Universidade Estadual de Campinas
Faculdade de Engenharia Elétrica e Computação
Departamento de Engenharia da Computação e Automação Industrial

Projeto de *caches* de matrizes particionados baseados em rastros de acesso à memória para sistemas embarcados

Autora: Marina Tachibana
Orientadora: Alice Maria Bastos Hubinger Tokarnia

Comissão Examinadora
Profa. Dra. Alice Maria Bastos Hubinger Tokarnia (FEEC/UNICAMP) - Presidente
Prof. Dr. Paulo Cesar Centoducatte (IC/UNICAMP)
Prof. Dr. Marco Aurélio Amaral Henriques (FEEC/UNICAMP)

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da UNICAMP, como parte dos requisitos exigidos para a obtenção do título de Mestre em Engenharia Elétrica. Área de concentração: Engenharia de Computação.

Campinas, SP - Brasil
29 de Abril de 2010

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA E ARQUITETURA - BAE - UNICAMP

T117p

Tachibana, Marina
Projeto de caches de matrizes particionados baseados em rastros de acesso à memória para sistemas embarcados / Marina Tachibana. --Campinas, SP: [s.n.], 2010.

Orientador: Alice Maria Bastos Hubinger Tokarnia.
Dissertação de Mestrado - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Memória cache. 2. Sistemas embutidos de computadores. I. Tokarnia, Alice Maria Bastos Hubinger. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Título em Inglês: Design of trace-based split array caches for embedded applications
Palavras-chave em Inglês: Split array caches, Embedded systems
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora: Paulo Cesar Centoducatte, Marco Aurélio Amaral Henriques
Data da defesa: 29/04/2010
Programa de Pós Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - TESE DE MESTRADO

Candidata: Marina Tachibana

Data da Defesa: 29 de abril de 2010

Título da Tese: "Projeto de Caches de Matrizes Particionados Baseados em Rastros de Acesso à Memória para Sistemas Embarcados"

Profa. Dra. Alice Maria Bastos Hubinger Tokarnia (Presidente): _____

Prof. Dr. Paulo César Centoducatte: _____

Prof. Dr. Marco Aurélio Amaral Henriques: _____

Agradecimentos

Em primeiro lugar, gostaria de agradecer a Deus, por ter me dado força, paciência e perseverança para conciliar minha vida acadêmica, profissional e pessoal, durante todo o período em que estive trabalhando nesta dissertação.

Gostaria também de agradecer a meus pais, Anita Kuniko Tachibana e Sekiya Tachibana, por sempre terem me incentivado a estudar, a me aprimorar como pessoa e a nunca desistir dos meus sonhos.

À minha irmã Miriam Tachibana, doutoranda em Psicologia pela PUC Campinas e Universidade de Lille, França, gostaria de agradecer seu apoio incondicional, assim como por ser uma fonte de inspiração e orgulho para mim, por sua paixão à vida acadêmica.

Ao meu marido Ricardo Ratti de Oliveira, agradeço todo o seu amor, companheirismo e compreensão.

Agradeço a minha orientadora Alice Tokarnia, que sempre acreditou que eu seria capaz de realizar um trabalho de mestrado, e com isso abriu minhas portas para a vida acadêmica, ensinando-me a pensar, criar e escrever.

À UNICAMP, Faculdade de Engenharia Elétrica e Computação e Laboratório de Computação e Automação Industrial, meus agradecimentos pela oportunidade que me deram como mestranda, assim como pela disponibilização de diversos computadores para o processamento em paralelo de minhas inúmeras simulações.

Aos membros da banca, Prof. Dr. Paulo Cesar Centoducatte e Prof. Dr. Marco Aurélio Amaral Henriques, agradeço pelos diversos comentários que contribuíram para melhorar o texto desta dissertação.

Também não poderia deixar de agradecer às empresas Motorola e Venturus, por terem investido tanto em mim, e em função disso terem me concedido 4 horas semanais durante o expediente de trabalho para eu me dedicar exclusivamente ao mestrado.

A todas minhas amigas, especialmente minhas amigas do colegial, Laura Silveira Moriyama, Sandra Sakanaka, Renata Niskacen e Cristiane de La Hoz, agradeço por me entenderem tão bem e por terem torcido tanto para que eu pudesse completar esta etapa tão importante da minha vida.

Marina Tachibana

"*Guarda siempre*
en tu corazón,
un lugar para
tus sueños..."

Autor anônimo, 2006

Resumo

Um sistema embarcado executa um único programa ou um conjunto pré-definido de programas repetidamente e, muitas vezes, seus componentes podem ser customizados para satisfazer uma especificação com requisitos referentes à área, desempenho e consumo de energia. *Caches on-chip*, em particular, são alvos de muitos algoritmos de customização por terem uma contribuição importante no desempenho e no consumo de energia de processadores embarcados.

Várias aplicações embarcadas processam estruturas de dados cujos padrões de acesso distintos tornam difícil encontrar uma configuração para o *cache* que garanta desempenho e baixo consumo. Propomos, neste trabalho, uma metodologia para projetar *caches* de matrizes particionados que satisfaçam uma restrição de tamanho total e em cujas partições estão mapeadas as matrizes da aplicação. Estas partições exploram a diferença de localidade espacial entre as matrizes. Com base na simulação de rastros de acesso à memória para entradas típicas, definimos uma métrica que quantifica o uso que as matrizes fazem das metades das linhas de um *cache* de matrizes unificado, associativo por conjunto, que satisfaz uma restrição de tamanho. Esta métrica é usada para dividir as matrizes em dois grupos, que são mapeados em duas partições de *cache*, uma com mesmo tamanho de linha, e outra com metade do tamanho de linha do *cache* de matrizes unificado. Este procedimento é repetido para várias organizações de *cache* de matrizes unificados com um tamanho especificado. No final, os *caches* de matrizes particionados baseados em rastros de acesso à memória com menor tempo médio de acesso à memória são selecionados.

Para um decodificador MPEG-2, dependendo do paralelismo dos acessos de dados, os resultados das simulações mostram que o tempo médio de acesso à memória de um *cache* de matrizes particionado baseado em rastros de 8K *bytes* apresenta uma redução de 26% a 60%, quando comparado com o *cache* de matrizes unificado, associativo por conjunto, de mesmo tamanho, com menor tempo médio de acesso à memória. Existe também uma redução de 46% no consumo de energia entre estes *caches*.

Abstract

An embedded system executes a single application or a pre-defined set of applications repeatedly and, frequently, its components can be fine-tuned to satisfy a specification with requirements related to area, performance, and energy consumption. On-chip caches, in particular, are the target of several customization algorithms due to its important contribution to the performance and energy consumption of embedded processors.

Several embedded applications process data structures whose access patterns turn it difficult to find a cache configuration that guarantees performance and low energy consumption. In this work, we propose a methodology for designing a split array cache that satisfies a total size constraint and in whose partitions the arrays of an application are mapped. Those partitions explore the difference in spatial locality among the matrices. Using traces of memory accesses, obtained for typical input patterns, we define a metric that quantifies the use of the two halves of the lines by array accesses in a unified array set-associative cache that satisfies a size constraint. We use this metric to split the arrays in two groups that are mapped to two cache partitions, one with the same line size, and the other with half line size of that of the unified array cache. This procedure is repeated for several unified array cache organizations of a specified size. In the end, the trace based split array caches with lowest average memory access time are selected.

For a MPEG-2 decoder, depending on the parallelism of array accesses, simulation results show that the average memory access time of an 8K byte split array cache is reduced from 26% to 60% as compared to that of the unified set associative array cache of same size with the lowest average memory access time. There is also a reduction of 46% in the consumption of energy.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

CACTI – Cache Access and Cycle Time Model

CAD – Computer Aided Design

CPU – Central Processing Unit

DRAM – Dynamic Random Access Memory

FFT – Fast Fourier Transform

FIFO – First In First Out

IDCT – Inverse Discrete Cosine Transform

ITU-T – International Telecommunication Union-Telecommunication

JPEG – Joint Photographic Experts Group

KB – Kilo bytes

LRU – Least Recently Used

MPEG – Moving Picture Experts Group

PC – Program Counter

RISC – Reduced Instruction Set Computer

ROM – Read Only Memory

SRAM – Static Random Access Memory

YUV – Luminance Bandwidth Chrominance

Corporações

ARC – Advanced RISC Computing

ARM – Advanced RISC Machine

DEC – Digital Equipment Corporation

MIPS – Microprocessor without Interlocked Pipeline Stages

SPEC – Standard Performance Evaluation Corporation

Trabalhos afins publicados pelo autor

Tokarnia, Alice. M., Tachibana, Marina, "*Design of Trace-Based Split Array Caches for Embedded Applications*", 13th Euromicro Conference on Digital System Design: Architecture, Methods, and Tools, Lille, France, 1-3 September 2010.

Capítulo 1

Introdução

Muitos sistemas embarcados diferem dos demais sistemas de propósito geral por serem reativos, ou seja, por permanecerem em modo de espera na maior parte do tempo, sendo ativados por eventos externos. Ao reagir a um evento, o sistema embarcado realiza um processamento em tempo real e pode gerar novos eventos. A corretude de um sistema embarcado é determinada não só pelo funcionamento adequado de seu algoritmo de computação, mas também pelo seu tempo de resposta, que deve ser adequado ao ambiente [1], [2]. Sistemas embarcados não podem deixar de atender seus prazos de execução, sendo preciso usar estimativas do tempo de execução de seus processos durante o desenvolvimento [5]. Outra diferença em relação aos sistemas de propósito geral, que executam diversos programas distintos ao longo de sua vida, é que sistemas embarcados muitas vezes executam um conjunto de programas fixos [8], residentes em ROM [9]. Alguns exemplos de sistemas embarcados são encontrados nos sistemas de supervisão e controle de tráfego aéreo, sistemas eletrônicos automotivos, telefones celulares e *tablets*.

O desenvolvimento da tecnologia de circuitos integrados tem possibilitado uma variedade de novas aplicações embarcadas, muitas das quais incluem acessos freqüentes a estruturas de dados. Nestas aplicações, os *caches* de dados localizados no mesmo *chip* que o processador são fundamentais para alcançar os requisitos de desempenho de tempo real, além de serem responsáveis por até 50% do total da energia consumida por um processador embarcado [6].

Este cenário transformou os *caches* em alvo de muitas pesquisas que visam otimizar o desempenho e o consumo de energia de sistemas embarcados. Algumas destas pesquisas têm como objetivo determinar, a partir dos padrões de acesso às variáveis de aplicações embarcadas, a

organização do *cache*, isto é, o tamanho de linha, a associatividade e o número de linhas. A possibilidade de customizar a organização do *cache* para uma aplicação em particular já pode ser encontrada em alguns processadores configuráveis, como MIPS [22], ARM [23], Arc [24], Tensilica Xtensa [25] e Nios [26], sendo este último da família de processadores embarcados da Altera.

Caches de matrizes são usados para vetores, matrizes e estruturas de dados complexas de uma aplicação embarcada. Neste trabalho, propomos uma metodologia para projetar *caches* de matrizes particionados que satisfaçam uma restrição de tamanho total e em cujas partições estão mapeadas as matrizes da aplicação. Usando simulação baseada em rastros de acesso à memória, particionamos as matrizes em dois grupos, conforme uma métrica relacionada à localidade espacial das matrizes. Cada grupo de matrizes é mapeado na partição C_1 ou C_2, onde a linha de C_1 tem o dobro do tamanho da linha de C_2.

Dado um tamanho de *cache*, este procedimento de particionamento é repetido para *caches* com diferentes tamanhos de linha *b*, associatividades *n* e conjuntos *m*. Um exemplo de *cache* particionado é mostrado na Figura 1.1.

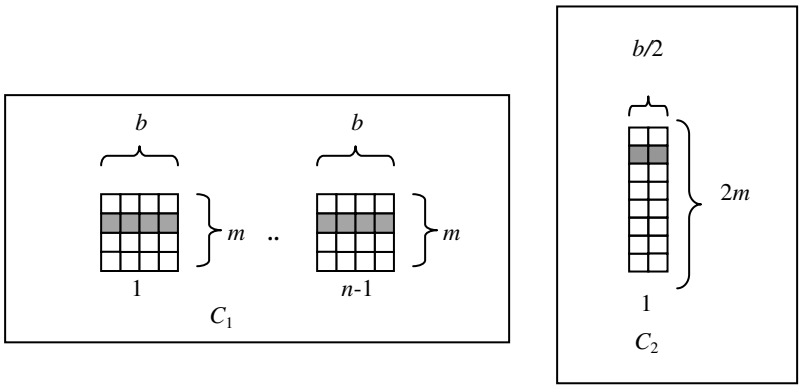


Figura 1.1 Exemplo de *cache* particionado.

Este procedimento pode ser aplicado iterativamente, gerando *caches* com múltiplas partições associadas a grupos menores de matrizes. Os resultados das nossas simulações para aplicações embarcadas mostram que estes *caches* podem reduzir o tempo médio de acesso à memória e o consumo de energia, quando comparados a *caches* de matrizes unificados associativos por conjunto.

A motivação para este trabalho foi a observação de que o tempo de acesso à memória não varia monotomicamente com o tamanho de linha e a associatividade para *caches* de um determinado

tamanho, conforme apresentado nas Figuras 1.2 e 1.3. Estas figuras correspondem a tabelas encontradas em [7] e correspondem a execuções de programas do SPEC 92 na estação DEC 5000.

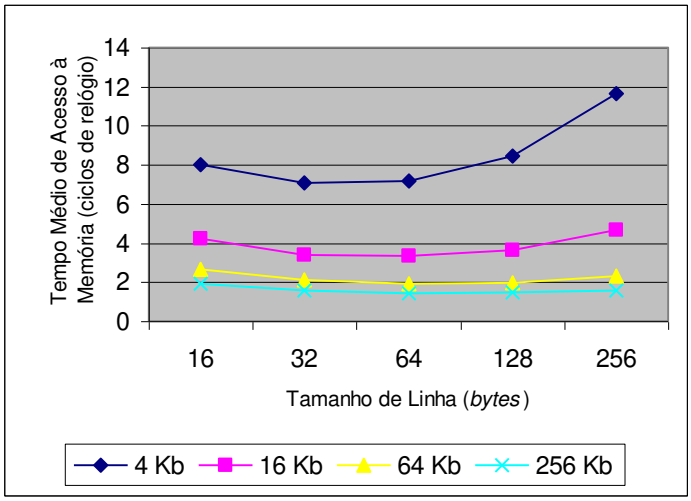


Figura 1.2 Tempo médio de acesso à memória como função do tamanho de linha para diferentes tamanhos de *cache* [7].

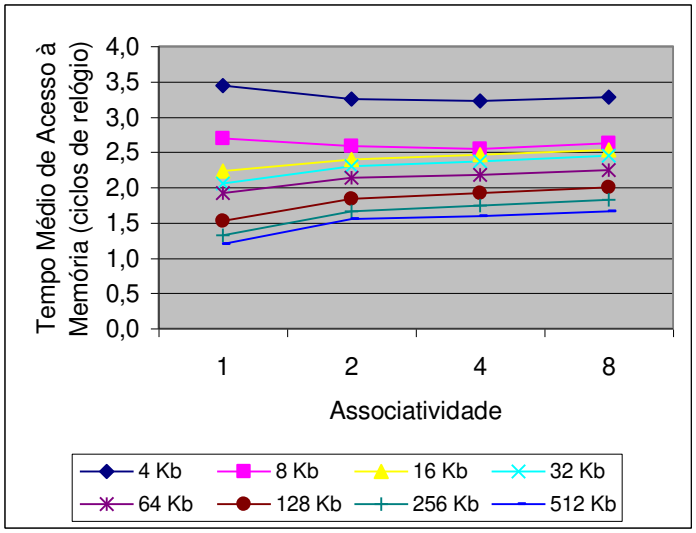


Figura 1.3 Tempo médio de acesso à memória como função da associatividade para diferentes tamanhos de *cache* [7].

De uma maneira geral, mantendo-se o mesmo tamanho de *cache* e associatividade, espera-se que aumentos no tamanho da linha reduzam o tempo médio de acesso à memória devido à localidade espacial. Entretanto, analisando a Figura 1.2, vemos que existe um tamanho de linha a partir do qual

aumentos no tamanho da linha resultam em aumentos no tempo médio de acesso à memória. Isto ocorre devido à redução no número de linhas no *cache*, que acarreta um aumento na taxa de falta.

De maneira análoga, para um mesmo tamanho de *cache* e de linha, quando a associatividade aumenta, a quantidade de blocos de memória mapeados para o mesmo conjunto de *cache* aumenta, reduzindo a competição pela linha de um mesmo conjunto. Desta forma, de um modo geral, espera-se uma diminuição no tempo médio de acesso à memória com o aumento da associatividade [7]. No entanto, a Figura 1.3 mostra que também existe uma associatividade a partir da qual não é mais possível reduzir o tempo médio de acesso à memória.

Nosso método procura por valores de tamanho de linha e associatividade que minimizam o tempo médio de acesso à memória do *cache* de matrizes unificado, que é decomposto em partições dedicadas de *cache* com organizações ajustadas aos padrões de acesso das variáveis mapeadas a cada partição.

Nossos *caches* particionados de matrizes, baseados em rastros, foram obtidos com o auxílio de um simulador de *cache* de matrizes associativo por conjunto, também desenvolvido neste trabalho. A partir de um rastro de acesso à memória, este simulador analisa a distribuição de acessos das linhas, registrando o número de acessos, faltas de *cache* e número de alternâncias de acessos às metades das linhas de *cache*. A avaliação da organização do *cache* é baseada em medidas da taxa de falta e em cálculos do tempo médio de acesso à memória e consumo de energia usando parâmetros baseados em modelos de *cache* e memória.

Esta dissertação está organizada em 6 capítulos, complementados por 9 apêndices.

O Capítulo 2 fornece uma visão geral da hierarquia de memória, de conceitos básicos referentes a *caches* e apresenta uma análise do impacto de *caches* sobre o desempenho do sistema. Algumas técnicas para melhorias de desempenho em *caches* são sucintamente descritas. O uso de *caches* em sistemas embarcados também é tratado neste capítulo.

O Capítulo 3 analisa alguns trabalhos publicados sobre otimizações na hierarquia de memória para sistemas embarcados.

O Capítulo 4 contém a descrição detalhada do algoritmo desenvolvido neste trabalho, considerando também sua aplicação iterativa. Apresentamos também algumas estruturas para a implementação de *caches* particionados.

O Capítulo 5 descreve os resultados experimentais obtidos através da utilização do algoritmo em cinco aplicações embarcadas.

O Capítulo 6 traz as contribuições deste trabalho e, também, sugestões para desenvolvimentos futuros.

O Apêndice I descreve o simulador de *cache* desenvolvido neste trabalho.

O Apêndice II traz os valores dos parâmetros das equações de tempo médio de acesso à memória e consumo de energia, utilizadas na avaliação dos *caches*.

O Apêndice III mostra, para cada matriz acessada nas aplicações embarcadas usadas para avaliação, o tamanho total e o tamanho adicional, devido à fragmentação de memória, das matrizes.

O Apêndice IV apresenta o número total de referências acessadas, o número total de linhas distintas acessadas e a taxa de faltas das aplicações embarcadas nos *caches* de matrizes unificados.

O Apêndice V apresenta as frações de alternâncias nos acessos às metades da linha e o número de blocos distintos acessados por cada matriz nas aplicações embarcadas nos *caches* de matrizes unificados.

O Apêndice VI apresenta todos os *caches* de matrizes particionados considerados para cada aplicação, com as métricas obtidas nas simulações e o tempo médio de acesso à memória.

O Apêndice VII apresenta, para cada aplicação, todos os *caches* de matrizes particionados considerados nas aplicações iterativas do algoritmo, com as métricas obtidas nas simulações e o tempo médio de acesso à memória.

O Apêndice VIII mostra as reduções no tempo médio de acesso à memória, consumo de energia e taxa de falta quando os *caches* de matrizes unificado e particionado com menor consumo de energia são comparados.

O Apêndice IX mostra as reduções no tempo médio de acesso à memória, consumo de energia e taxa de falta quando os *caches* de matrizes unificado e particionado com menor produto do consumo de energia e tempo médio de acesso à memória são comparados.

Capítulo 2

Caches : conceitos básicos

2.1 Introdução

Este capítulo tem como objetivo introduzir conceitos relativos à hierarquia de memória, em especial, aos *caches*. Uma análise quantitativa do impacto de *caches* sobre o desempenho do sistema é apresentada. As principais técnicas para melhorias de desempenho de *caches* são também comentadas. O capítulo termina descrevendo alguns *caches* usados em sistemas embarcados.

2.2 Hierarquia de Memória

A hierarquia de memória é um mecanismo empregado para que o processador possa operar como se dispusesse de grande quantidade de memória com baixo tempo de acesso e baixo custo por *bit* armazenado. Diversos níveis de memória compõem uma hierarquia de memória. Como ilustrado na Figura 2.1, cada nível de memória é menor, mais rápido e mais caro por *bit* conforme sua distância do processador aumenta.

Uma organização bastante difundida é a hierarquia de *cache* de múltiplos níveis. Como podemos ver na Figura 2.2, esta hierarquia de memória é composta por um *cache* nível 1, pequeno e de baixa latência, e por um *cache* nível 2, maior e mais lento, e pela memória principal [12]. O *cache* nível 1 é comumente particionado em *cache* de dados e de instruções, ao passo que o *cache* nível 2 é unificado [13].

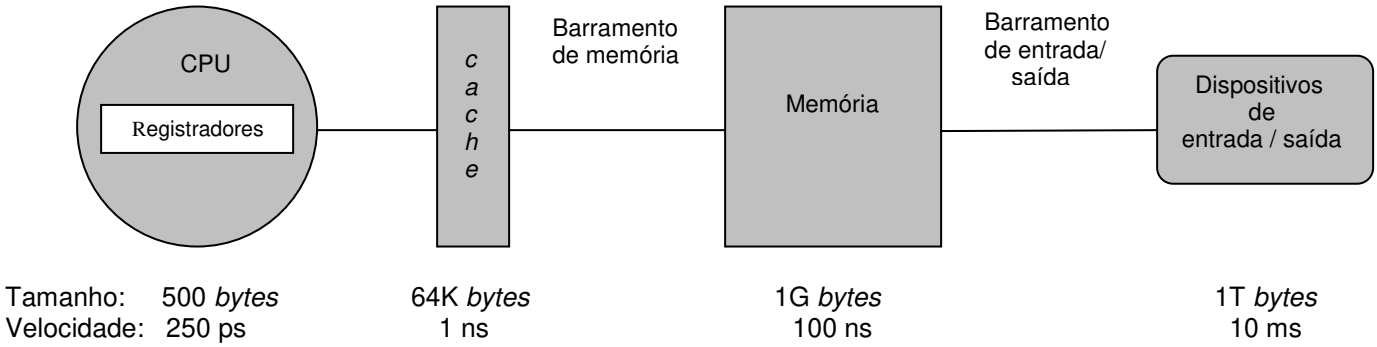


Figura 2.1 Exemplo de níveis de hierarquia de memória [7].

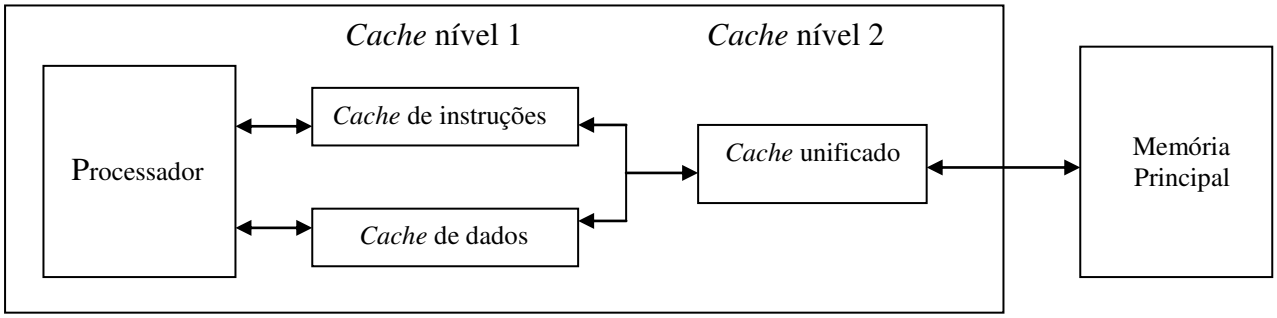


Figura 2.2 Hierarquia de *cache* de dois níveis [13].

A crescente importância da hierarquia de memória pode ser observada na Figura 2.3, que mostra que o desempenho dos processadores cresceu cerca de dez mil vezes nos últimos vinte e cinco anos, ao passo que o da memória cresceu menos de dez vezes no mesmo período. Visto que o processador precisa acessar a memória para operar, o fato do desempenho da memória ter crescido aproximadamente mil vezes menos que o do processador impõe uma barreira para que os ganhos de desempenho conseguidos no processador resultem em ganhos no desempenho do sistema, aumentando a importância de uma hierarquia de memória eficiente.

Nas próximas subseções explicamos alguns conceitos relativos a *caches*, detalhamos o mapeamento da linha de *cache* a partir do endereço do dado, classificamos os tipos de falta no *cache* e mencionamos problemas típicos encontrados nos *caches*. A seção é finalizada com um exemplo de funcionamento do *cache*.

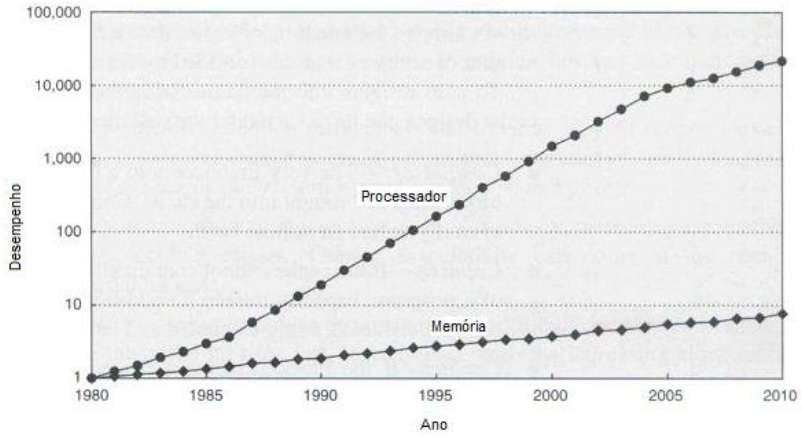


Figura 2.3 Histórico e projeção da diferença de crescimento no desempenho entre memória e processador [7].

2.2.1 Conceitos Relativos à *Caches*

O *cache* corresponde ao primeiro nível de hierarquia de memória, o mais próximo do processador, conforme vemos na Figura 2.1. Quando o processador busca um dado no *cache*, se este dado é encontrado no *cache*, dizemos que houve um acerto (*hit*) no *cache*. Senão, dizemos que houve uma falta (*miss*) no *cache*.

Como a localidade temporal diz que é provável que uma *word* requisitada seja referenciada novamente no futuro próximo, esta *word* é trazida da memória principal para o *cache*. Assim, se ela for referenciada novamente, o próximo acesso será mais rápido, uma vez que o acesso ao *cache* é mais rápido que à memória principal.

Além disso, não é somente a *word* requisitada pelo processador que é colocada no *cache*, mas uma linha de *cache*, que consiste num conjunto de dados de tamanho fixo que contém a *word* referenciada. Os dados vizinhos a *word* requisitada são também trazidos ao *cache* para se tirar proveito da localidade espacial, segundo a qual dados com endereços próximos ao do dado acessado apresentam uma alta probabilidade de serem acessados num futuro próximo.

O *cache* é composto por *m* conjuntos de *n* linhas de tamanho *b bytes*. A quantidade *n* de linhas por conjunto, também denominada quantidade de *vias*, define a *associatividade* do *cache*. Os valores de *b*, *n* e *m* definem a configuração e tamanho do *cache*.

A Figura 2.4 ilustra um exemplo de organização de *cache*, onde as linhas de um mesmo conjunto, mas de vias diferentes, estão sombreadas.

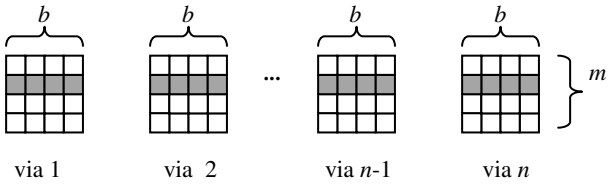


Figura 2.4 Exemplo de organização de *cache*.

O tamanho total *c bytes* do *cache* é dado por:

$$c = b \times n \times m \qquad \text{(eq. 2.1)}$$

Como o modo de operação do *cache* não interfere no funcionamento do programa, mas somente no seu desempenho [9], um mesmo programa pode ser executado em diferentes configurações de *cache*.

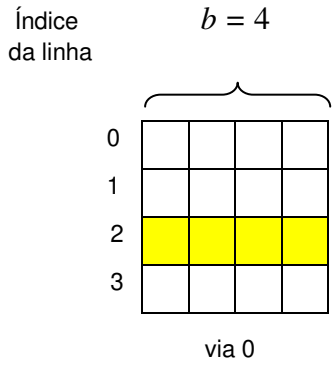
Um bloco de memória pode ser mapeado em qualquer linha de *cache* do conjunto determinado pela equação a seguir:

$$conjunto\_de\_linha\_de\_cache = (endereço\_bloco\_de\_memória)\ MOD\ (m) \qquad \text{(eq. 2.2)}$$

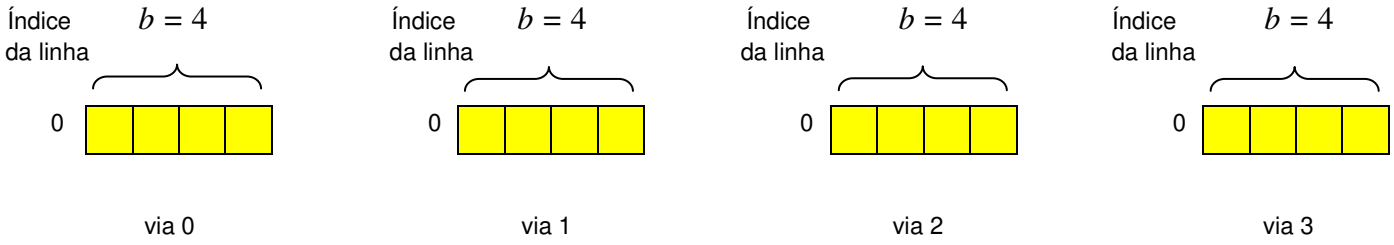
Conforme mostrado na Figura 2.5, o valor de *n* determina a estratégia de mapeamento do *cache*. Para $n = 1$, o *cache* é diretamente mapeado, e cada bloco de memória pode ser mapeado em apenas uma linha de *cache*. Por outro lado, para $n = número\_de\_linhas\_no\_cache$, o *cache* é completamente associativo e um bloco de memória pode ser armazenado em qualquer linha no *cache*. Para valores de *n* entre 1 e *número_de_linhas_no_cache*, o *cache* é associativo por conjunto e um bloco de memória pode ser armazenado em qualquer uma das *n* linhas de *cache* que pertence ao conjunto específico de linhas determinado pela equação 2.2.

Depois que o conjunto no qual o bloco é mapeado é determinado, para os *caches* associativos falta ainda determinar que linha neste conjunto deve receber este bloco. Esta especificação depende da estratégia de substituição de blocos adotada pelo *cach*e. Pela estratégia aleatória, o bloco é selecionado aleatoriamente para esta substituição, de modo que, na média, os blocos escolhidos para serem substituídos são distribuídos uniformemente pelo conjunto dos blocos. Na estratégia *LRU* (*Least*

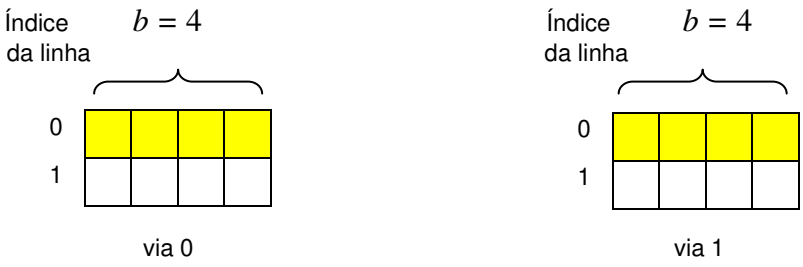
Recently Used), o bloco menos recentemente usado é escolhido para substituição. Esta estratégia armazena o histórico de acesso de cada linha do *cache* e, baseada na propriedade de localidade temporal, assume que os blocos mais recentemente usados serão provavelmente utilizados novamente. Como a estratégia *LRU* implica em estruturas e tempo de cálculo, uma estratégia de substituição mais simples e rápida que também é aplicada é a *FIFO* (*First In, First Out*). Por esta estratégia, o primeiro bloco armazenado no *cache* é o bloco a ser substituído, de forma que o bloco de maior tempo de permanência no *cache* é descartado.



(a) Diretamente mapeado: bloco 6 pode ser mapeado somente para a linha 2 (6 *mod* 4).



(b) Completamente associativo: bloco 6 pode ser mapeado para qualquer linha.



(c) Associativo por conjunto: bloco 6 pode ser mapeado em qualquer linha do conjunto 0 (6 *mod* 2).

Figura 2.5 Mapeamento do bloco 6 de acordo com cada estratégia de mapeamento de *cache* [7].

Quando o bloco é copiado da memória principal para o *cache*, passam a existir duas instâncias do bloco, uma localizada na memória e outra, no *cache*. Assim, quando há um acesso de escrita no bloco, é necessário usar uma estratégia de escrita para gerenciar ambas as instâncias do dado. Pela estratégia de escrita através (*Write Through*), ambas as instâncias são atualizadas. Já pela estratégia de escrita na volta (*Write Back*), a escrita é feita somente no bloco no *cache*. A memória principal é atualizada só quando o bloco de *cache* correspondente for retirado do *cache*. Neste caso, com o intuito de evitar atualizações desnecessárias da memória principal quando um bloco é substituído do *cache*, é comum utilizar-se o *bit* de escrita (*dirty bit*) para indicar se o bloco foi escrito enquanto estava no *cache*. Deste modo, a memória principal é atualizada somente se este *bit* sinalizar que houve modificações no bloco durante sua permanência no *cache*. Além do *bit* de escrita, os *caches* utilizam o *bit* de validade (*valid bit*) presente em cada linha de *cache* para identificar se o dado armazenado na linha é válido.

Caches podem abrigar tanto dados quanto instruções. No entanto, como os processadores podem requisitar dados e instruções no mesmo ciclo de relógio, o uso de um *cache* unificado pode causar gargalos. Para evitá-los, o *cache* pode ser particionado em dois *caches*, um para dados, e outro para instruções. Além de duplicar a largura de banda entre hierarquia de memória e processador, este particionamento remove as faltas no *cache* unificado ocasionadas pela competição entre dados e instruções pela mesma linha de *cache*. Entretanto, se por um lado, este particionamento permite que a configuração seja otimizada individualmente para cada *cache*, por outro lado, ela torna fixo o tamanho do *cache* de dados e instruções, impedindo-os de ter a flexibilidade de usar mais ou menos espaço de *cache*.

2.2.2 Mapeamento de Linha de *Cache* Baseado no Endereço do Dado

A identificação das linhas onde o endereço pode estar armazenado em um *cache* associativo por conjunto ou diretamente mapeado é feita a partir do endereço fornecido pelo processador e da etiqueta (*tag*) de endereço da linha que cada linha do *cache* armazena. Conforme ilustrado na Figura 2.6, o endereço do dado requisitado é composto por três partes. Enquanto o índice seleciona o conjunto da linha, a etiqueta identifica a linha e o deslocamento na linha indica a posição do endereço dentro da linha.

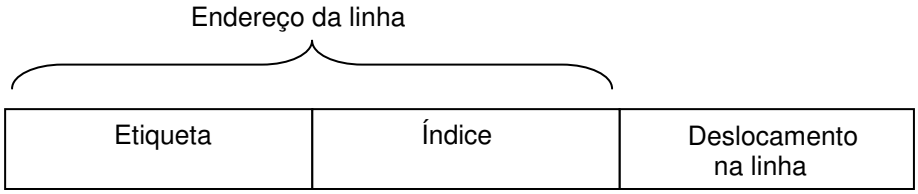


Figura 2.6 Partes de um endereço em um *cache* associativo por conjunto ou diretamente mapeado [7].

Primeiramente, separam-se os *bits* do endereço do dado entre os *bits* que indicam o endereço da linha e o deslocamento (*offset*) na linha aplicando-se as Equações (2.3) e (2.4).

$$número\_bits\_do\_deslocamento\_na\_linha = \log_2 b \qquad \text{(eq. 2.3)}$$

$$número\_bits\_do\_endereço\_da\_linha = número\_bits\_do\_endereço - número\_bits\_do\_deslocamento\_na\_linha \qquad \text{(eq. 2.4)}$$

Em seguida, divide-se o endereço do bloco entre os campos índice e etiqueta com base nas Equações (2.5) e (2.6). Quando o dado é armazenado no *cache*, é este campo etiqueta do endereço que é armazenado no campo etiqueta da linha de *cache*.

$$número\_bits\_do\_índice\_da\_linha = \log_2 m = \log_2\left(\frac{c}{b \times n}\right) \qquad \text{(eq. 2.5)}$$

$$número\_bits\_da\_etiqueta\_do\_bloco = número\_bits\_do\_endereço\_da\_linha - número\_bits\_do\_índice\_da\_linha \qquad \text{(eq. 2.6)}$$

Para verificar se o dado requisitado está no *cache*, compara-se o valor do campo etiqueta do endereço requisitado com a etiqueta de cada linha válida do conjunto no qual o bloco deve ser armazenado. Caso um deles seja igual, há um acerto no *cache*.

Em casos de *caches* associativos por conjunto de *n* vias, como há *n* linhas num mesmo conjunto, é preciso checar a etiqueta de todas as *n* linhas do conjunto identificado pelo campo índice, pois qualquer um deles pode conter o bloco endereçado.

2.2.3 Causas de Faltas no *Cache*

Existem três razões pelas quais as faltas no *cache* ocorrem.

Primeiramente, há a falta compulsória, que é a falta que não pode ser evitada porque se refere à primeira vez que o bloco é acessado.

Já a falta por capacidade está relacionada com a falta de espaço no *cache* para armazenar todos os blocos acessados, de modo que blocos descartados para dar espaço a novos blocos serão levados novamente ao *cache* quando acessados no futuro.

O último motivo é aplicável somente a *caches* diretamente mapeados ou associativos por conjunto, onde as faltas de conflito acontecem se vários blocos são mapeados para o mesmo conjunto de linhas, causando o descarte de um bloco que mais tarde precisará ser trazido ao *cache* novamente.

2.2.4 Problemas Encontrados em *Caches*

Apesar dos *caches* terem o objetivo de melhorar o desempenho do sistema, em alguns casos eles podem até piorá-lo.

Como o *cache* trata todas as referências igualmente, sem distinção de tipo de localidade, ou da falta dela, o seu desempenho pode ser comprometido para alguns padrões de acesso. Por exemplo, quando as referências vizinhas à referência acessada são repetidamente levadas ao *cache* sem serem posteriormente acessadas, dizemos que o *cache* está poluído de referências que são trazidas ao *cache* devido ao acesso a uma referência com baixa localidade espacial.

Além disso, a forma de mapeamento dos blocos em *caches* associativos por conjunto pode mapear para a mesma linha de *cache* referências acessadas no mesmo trecho de código. Esta competição pelas mesmas linhas de *cache* em instantes de tempo próximos obriga as referências conflitantes a se excluírem mutuamente do *cache*, acarretando faltas no *cache* mesmo quando as outras linhas estão livres [12].

2.2.5 Exemplo de Funcionamento do *Cache*

Nesta seção, apresentamos um exemplo simples de funcionamento do *cache* para o rastro de acesso à memória e configuração de *cache* representados nas Figuras 2.7 e 2.8, respectivamente.

Supomos que a matriz *M* armazena valores inteiros, que possuem o mesmo número de *bytes* que uma *word*, isto é, 4 *bytes*, no nosso exemplo, e que a posição *M*[0] corresponde ao endereço 0x0012A764 (igual a $00000000100101010011101100100_2$), enquanto a posição *M*[1] corresponde ao endereço 0x0012A765 (igual a $00000000100101010011101100101_2$), e assim por diante.

acesso a *M*[0]

acesso a *M*[3]

Figura 2.7 Exemplo de um rastro de acesso à memória, em que a matriz *M* armazena tipos inteiros.

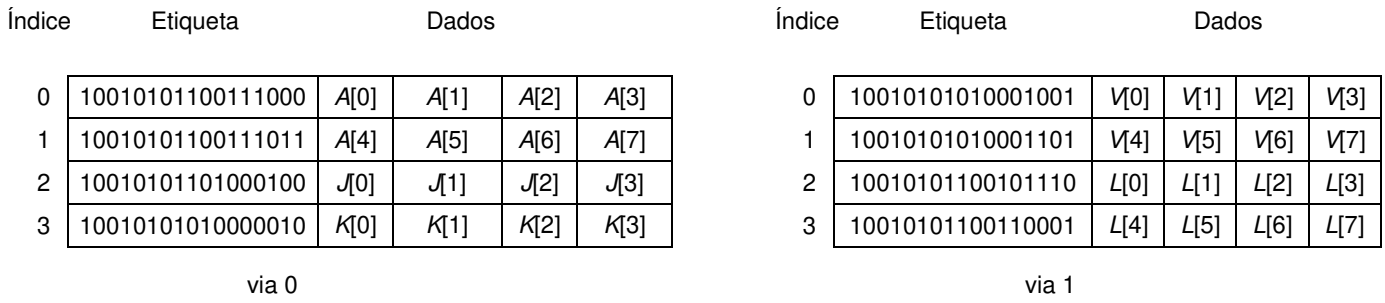


Figura 2.8 Exemplo de *cache* associativo por conjunto com *b*=4, *n*=2 e *m*=4.

Primeiramente, o endereço de *M*[0] é decomposto para identificar em qual linha de *cache* ele é mapeado. Aplicando-se a Equação 2.3, como $b = 4$ e $2^2 = 4$, sabemos que dois *bits* indicam o deslocamento na linha. Pela Equação 2.5, descobrimos que dois *bits* indicam o índice da linha, pois $32/(4 \times 2) = 4 = 2^2$. Na Figura 2.9 vemos que *M*[0] deve ser mapeado para a posição 0 da linha 1.

Na Figura 2.8, vemos que nenhuma das linhas com índice 1 tem etiqueta igual a 0x12A76 (10010101001110110_2), caracterizando uma falta no *cache*. Supondo que o bloco da via 1 seja o menos recentemente utilizado, ele é escolhido para *M*[0] e seus vizinhos, como na Figura 2.10.

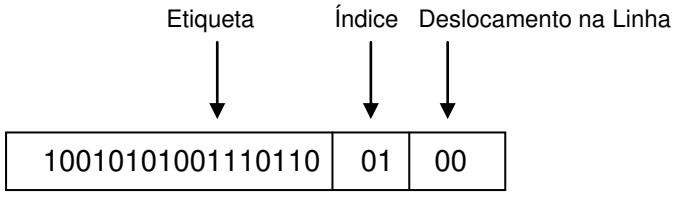


Figura 2.9 Decomposição do endereço de *M*[0]. Etiqueta 10010101001110110_2 é igual a 0x12A76.

Em seguida, pelo rastro da Figura 2.7, *M*[3] é acessado. Porém, como sua etiqueta 0x12A76 (igual a 10010101001110110_2) se encontra na via 1 do conjunto 1, temos um acerto no *cache*.

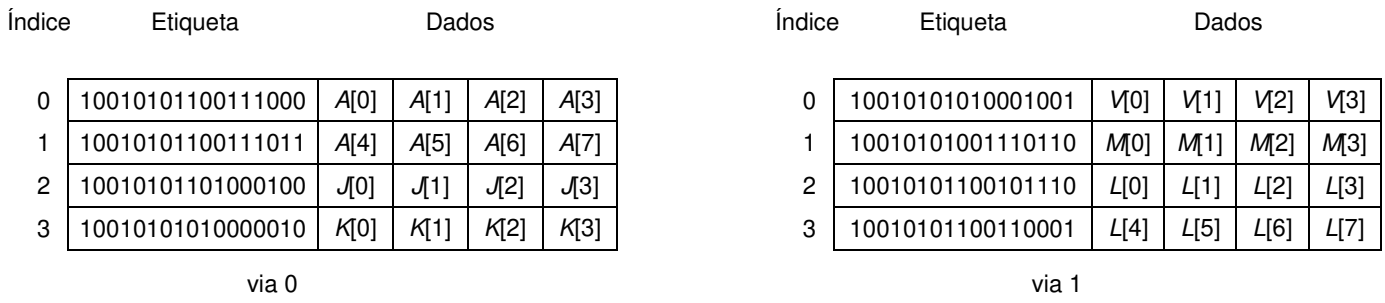


Figura 2.10 *Cache* armazenando *M*[0].

2.3 Análise Quantitativa do Impacto dos *Caches* no Desempenho do Sistema [7]

Considerando que parte dos acessos à hierarquia de memória pode resultar em acertos no *cache*, ao passo que a outra parte pode resultar em faltas, a avaliação do desempenho da hierarquia de memória analisa estas duas situações.

Para avaliar o impacto da fração de acessos que foi encontrada no *cache*, utilizamos o tempo de acerto, isto é, o tempo para acessar o *cache*. Por sua vez, o impacto da fração de acessos que não foi encontrada no *cache* é calculado pelo produto da taxa de faltas no *cache* pela penalidade da falta. A taxa de faltas fornece a percentagem dos acessos ao *cache* que ocasionaram faltas e a penalidade da falta indica o tempo médio adicional necessário para trazer o dado ao *cache*. Desta maneira, o desempenho da hierarquia de memória é avaliado pela métrica tempo médio de acesso à memória:

$$tempo\_médio\_de\_acesso\_à\_memória = tempo\_de\_acerto + taxa\_de\_falta \times penalidade\_da\_falta \quad \text{(eq. 2.7)}$$

Embora o tempo médio de acesso à memória não substitua a métrica de tempo de execução da aplicação, ela é uma métrica mais abrangente do que a taxa de faltas, pois considera o desempenho da hierarquia de memória.

De forma análoga, a energia consumida pode ser calculada a partir da energia no acerto e a penalidade de energia.

Enquanto a energia no acerto especifica a energia gasta para acessar o *cache*, a penalidade de energia determina a energia adicional consumida para trazer o dado ao *cache*. A energia adicional gasta quando há uma falta no *cache* é calculada pelo produto da taxa de falta pela penalidade de energia:

$$energia\_média\_consumida = energia\_no\_acerto + taxa\_de\_falta \times penalidade\_de\_energia \quad \text{(eq. 2.8)}$$

2.4 Técnicas para Melhoria de Desempenho do *Cache* [7]

Conforme vimos na seção anterior, o tempo médio de acesso à memória depende do tempo de acerto, taxa de falta e penalidade da falta. Portanto, otimizações em alguma dessas métricas reduzem o tempo médio de acesso à memória. No entanto, como as otimizações em uma desta métricas podem causar pioras em outras, uma solução de compromisso precisa ser avaliada.

Nas seções seguintes apresentamos algumas técnicas para melhoria de desempenho do *cache*, agrupadas pelo fator específico otimizado e com foco em *caches* de dados, pois nosso trabalho visa melhorar o desempenho de *caches* de dados.

2.4.1 Técnicas para Redução no Tempo de Acerto

A redução no tempo de acerto pode ser alcançada através do uso de *caches* pequenos e simples e da previsão de via.

Caches pequenos podem diminuir o tempo de acerto porque *hardwares* menores podem ser mais rápidos. Por sua vez, *caches* simples, como os diretamente mapeados, por exemplo, requerem menos passos para verificar se o dado está no *cache*.

A técnica de previsão de via acrescenta *bits* em cada conjunto do *cache* para prever a via ou linha deste conjunto que será acessada. Antes do acesso, as previsões das linhas que serão utilizadas são armazenadas nestes *bits*. Se a previsão for correta, o tempo de acerto é reduzido porque a comparação da etiqueta é conduzida no mesmo ciclo de relógio que o dado do *cache* é lido. Senão, outras linhas do conjunto são verificadas nos próximos ciclos de relógio.

2.4.2 Técnicas para Redução na Penalidade da Falta

Entre as técnicas existentes para a redução na penalidade da falta encontram-se as otimizações de reinício antecipado (*Early Restart*), *word* crítica primeiro (*Critical Word First*) e utilização de *buffers* de escrita.

Uma falta no *cache* ocorre quando uma *word* requisitada pelo processador não foi encontrada no *cache*. Embora somente uma *word* tenha sido requisitada, tanto a *word* requisitada quanto suas *words* vizinhas são trazidas ao *cache*. O *cache* só retorna a *word* requisitada quando a linha foi completamente preenchida no *cache*, forçando o processador a esperar por este preenchimento para retomar sua execução. Entretanto, como o processador geralmente necessita só de uma *word* por vez, uma forma de reduzir a penalidade da falta é disponibilizar a *word* para o processador assim que ela seja armazenada no *cache*, sem esperar pelas demais *words* da linha.

Pela estratégia do reinício antecipado, as *words* são preenchidas na linha de *cache* seguindo sua posição na linha. Porém, tão logo a *word* requisitada chega à linha, ela é repassada para o processador. Assim, o processador consegue prosseguir com sua execução ao mesmo tempo em que o resto da linha do *cache* é preenchida. A estratégia da *word* crítica primeira adota a mesma idéia que a estratégia do reinício antecipado, mas não obedece a ordem da posição da linha para requisitar as *words*, pedindo para a memória primeiro a *word* que ocasionou a falta. Deste modo, a *word* que ocasionou a falta é sempre a primeira a chegar ao *cache*. Como a redução da penalidade da falta está ligada ao tempo que o processador consegue executar em paralelo com o preenchimento do resto da linha do *cache*, quanto maior o tamanho da linha do *cache*, maior a redução na penalidade da falta.

A utilização de *buffers* de escrita reduz a penalidade da falta impedindo que o *cache* espere pela atualização na memória dos blocos que foram modificados no *cache*. Quando ocorre um acesso de escrita no *cache*, o dado é primeiro atualizado no *buffer* de escrita para só depois ser enviado para o próximo nível da hierarquia de memória, isto é, o nível seguinte em distância do processador. Uma vez que o dado é escrito no *buffer*, do ponto do vista do processador, a escrita foi finalizada e ele está pronto para retomar suas operações enquanto um *hardware* dedicado se encarrega de enviar o conteúdo do *buffer* para a memória. Os *buffers* de escrita são utilizados nas atualizações no bloco de memória.

2.4.3 Técnicas para Redução na Taxa de Falta

Para reduzir a taxa de falta, é comum aumentar o tamanho de linha, aumentar o grau de associatividade, utilizar *caches* pseudo-associativos, *caches* de vítimas, mecanismo de *prefetch* e algumas otimizações realizadas pelo compilador.

O aumento no tamanho de linha favorece a localidade espacial e reduz a taxa de faltas compulsórias. Porém, linhas maiores também podem aumentar a taxa de faltas compulsórias e a taxa de faltas por conflito, devido à redução no número de linhas. Além disso, linhas maiores têm uma penalidade de falta mais alta, pois um número maior de *words* deve ser trazido ao *cache*.

O aumento no grau de associatividade, por sua vez, reduz a taxa de faltas por conflito, por adicionar mais linhas a um conjunto. Por outro lado, o tempo de acerto pode ser elevado. Como mais linhas podem armazenar o dado requisitado, é necessário procurar o dado em mais linhas.

Os *caches* pseudo-associativos operam como *caches* diretamente mapeados quando há acerto no *cache*. Quando há falta, antes de consultar o próximo nível da hierarquia de memória, procura-se o dado numa outra entrada do *cache*, que seria o pseudoconjunto. Um procedimento simples para mapear dados num *cache* pseudo-associativo com 2 conjuntos consiste em inverter o *bit* mais significativo do campo índice da linha.

Os *caches* de vítimas são *caches* pequenos completamente associativos que armazenam somente as linhas descartados pelo *cache* durante uma falta. Diante de uma falta no *cache*, procura-se no *cache* de vítimas o bloco requisitado. Caso ele esteja lá, ele é trazido ao *cache*, ao passo que o bloco substituído é levado ao *cache* de vítimas. Caso o bloco não se encontre no *cache* de vítimas, ele é procurado no próximo nível da hierarquia de memória.

As otimizações feitas pelo compilador reduzem a taxa de falta sem precisar alterar o *hardware*. Entre as otimizações mais comuns, encontra-se a reordenação de código e dados.

Como a ordem de acesso ao *cache* é ditada pela ordem em que dados e instruções aparecem no programa, é possível reduzir as faltas por conflito alterando esta ordem. Em linhas longas, a quantidade de faltas pode ser diminuída posicionado a instrução com mais chance de ser executada logo após a instrução da condição e, em casos de códigos seqüenciais, alinhando no início da linha de *cache* a porção do bloco que constitui o ponto de entrada de acesso à linha.

No código da Figura 2.11 (a), a matriz *X* é armazenada por linha e a memória é acessada em intervalos de 100 *words*. Após trocar a ordem de *loops*, o código da Figura 2.11 (b), faz com que os

acessos sejam sequenciais sobre os dados de uma linha de *cache*. Ao melhorar a localidade espacial e temporal, esta técnica reduz a taxa de falta.

```
for (j = 0; j < 100; j = j + 1)
    for (i = 0; i < 5000; i = i + 1)
        X[i] [j] = 2 * X[i] [j] ;
```
(a)

```
for (i = 0; i < 5000; i = i + 1)
    for (j = 0; j < 100; j = j + 1)
        X[i] [j] = 2 * X [i] [j] ;
```
(b)

Figura 2.11 Código sem (a) e com (b) a otimização de troca de *loops* [7].

A redução da penalidade e da taxa de falta via paralelismo pode ser obtida pelo *prefetch*, executado pelo *hardware* ou por instruções introduzidas pelo compilador.

Um mecanismo muito utilizado pelos processadores é o *prefetch* de instruções controlado por *hardware*, que lê a instrução de memória antes que seja necessária sua execução. Quando ocorre uma falta de instrução, o processador busca o bloco requisitado e o seu bloco consecutivo, pois a chance do bloco consecutivo ser acessado é alta, segundo a propriedade da localidade espacial. Enquanto o bloco requisitado é armazenado no *cache* de instruções, o bloco adicional buscado pode ser armazenado ou no próprio *cache* ou num *buffer* de instruções. Quando a instrução adicional buscada é armazenada no *buffer*, o *prefetch* de instrução diminui a penalidade de falta, uma vez que o acesso ao *cache* ou ao *buffer* é mais rápido que à memória principal. Em particular, quando a instrução adicional é armazenada no *cache*, o *prefetch* de instrução diminui a taxa de falta, visto que os acessos a estas instruções constituiriam faltas no *cache* se não houvesse o *prefetch*.

Os benefícios deste mecanismo podem ser ampliados aplicando-se o *prefetch* de *hardware* também nos acessos a dados ou utilizando-se um *prefetch* controlado pelo compilador, no qual o compilador é responsável por adicionar as instruções de *prefetch* para evitar *prefetches* desnecessários ou desperdiçados. Para isso, devido à capacidade limitada do *cache*, o compilador se concentra no *prefetch* de referências que provavelmente resultarão em faltas de *cache* e evita tanto *prefetches* de dados já presentes no *cache* quanto *prefetches* que não conseguem trazer o dado antes dele ser necessário.

2.5 Memória em Sistemas Embarcados

Normalmente, os dados dos sistemas embarcados são armazenados ou na memória de acesso randômico estática (*SRAM*) ou na dinâmica (*DRAM*), sendo que as memórias *DRAM* compõem a memória principal.

A memória *scratch-pad* consiste num tipo de memória *SRAM*, mas operada exclusivamente como um vetor mapeado num endereço fixo. Para utilizá-la, é necessário que o código esteja customizado para um tamanho de memória *scratch-pad* [27]. Caso o tamanho desta memória mude, é preciso mudar também o código que a utiliza.

Os *caches* nível 1 são um tipo de memória *SRAM*, usualmente integrados no mesmo *chip* que o processador, possibilitando um acesso rápido.

Os programas presentes em sistemas embarcados gastam grande parte do seu tempo de execução em pequenos *loops* [10]. Tendo em vista que uma parte significativa do consumo de energia do sistema é referente a buscas por instruções no *cache* ou na memória principal, seja dentro ou fora do *chip*, diversas abordagens armazenam *loops* pequenos em um *cache* de *loops* pequeno e de baixo consumo de energia [8].

Conforme mostrado na Figura 2.12, o *cache* de *loops* é composto por um pequeno *buffer* de instruções e um controlador [11].

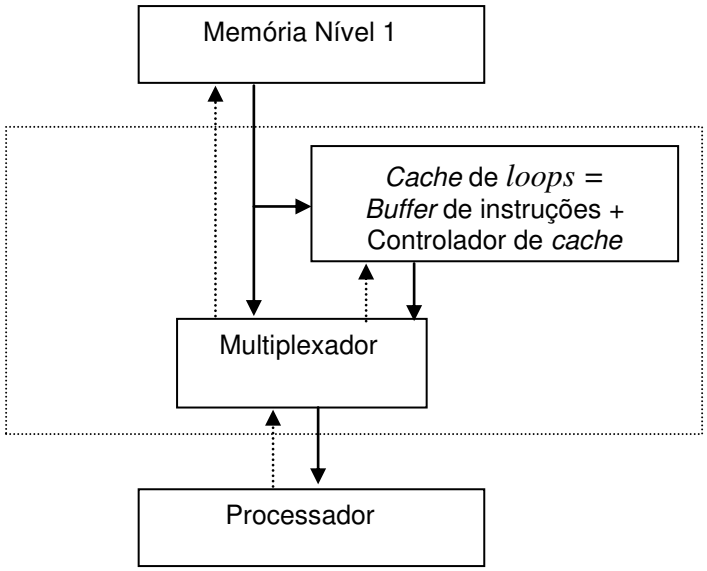


Figura 2.12 Arquitetura de um *cache* de *loops* [11].

Ao término da primeira iteração do *loop*, o controlador de *cache* de *loop* consegue detectar que a instrução final é referente a um *loop* simples. Já na segunda iteração do *loop*, ele preenche o *buffer* de instruções com as instruções do corpo do *loop*. A partir da terceira iteração, o controlador faz com que o multiplexador leia as instruções do *cache* de *loops* ao invés da memória [11]. Durante a execução de um *loop*, apenas as instruções deste *loop* são armazenadas. O *cache* de *loops* não necessita de etiqueta de endereço nem de *bit* de validade. Supondo que o *buffer* de instruções tenha capacidade suficiente para armazenar todo o *loop*, como este *buffer* é diretamente mapeado, as instruções presentes nos *loops* do programa são sempre mapeadas para um mesmo local. Desta forma, não há competição pelo mesmo local do *cache* entre instruções de um mesmo *loop* do programa [10].

2.6 Conclusão

Este capítulo introduziu conceitos relativos a *caches*, com ênfase nos conceitos mais importantes para a compreensão do nosso trabalho. Nosso objetivo é particionar um *cache* de matrizes unificado associativo por conjunto de n vias com tamanho de linha b em um *cache* associativo por conjunto de n_1 vias com tamanho de linha b e um cache associativo por conjunto de n_2 vias com tamanho de linha $b/2$, de modo que $n_1 + n_2 = n$ e o tempo médio de acesso à memória seja reduzido. Tanto o *cache* unificado quanto o particionado são *caches* nível 1 de matrizes que seguem a política do bloco menos recentemente utilizado para substituição de blocos.

As métricas introduzidas para avaliar o impacto dos caches foram: taxa de falta, tempo de acerto, penalidade de falta, tempo médio de acesso à memória, energia no acerto, penalidade de energia e energia média consumida. Nosso trabalho utiliza as métricas tempo médio de acesso à memória e consumo de energia porque elas abrangem as outras métricas. Elas são utilizadas neste trabalho para comparar *caches* de matrizes, inclusive *caches* de matrizes particionados definidos neste trabalho.

Visto que este trabalho tem como objetivo melhorar o desempenho do *cache*, este capítulo apresentou algumas técnicas bem conhecidas que possuem o mesmo objetivo, sendo que algumas delas são empregadas em processadores comerciais.

Capítulo 3

Trabalhos Anteriores

3.1 Introdução

Este capítulo tem a finalidade de descrever algumas das técnicas encontradas na literatura para melhorar o desempenho e consumo de energia de *caches* embarcados. Assim como o trabalho desenvolvido nesta dissertação, algumas destas técnicas ajustam a configuração do *cache* para um determinado conjunto de aplicações e mapeiam as matrizes em *caches* particionados conforme sua localidade.

3.2 Trabalhos Anteriores

Os trabalhos anteriores podem ser classificados em diversas categorias. Primeiramente, podemos dividir os trabalhos entre os que só se aplicam a *caches* de dados e os que se aplicam tanto a *caches* de dados quanto de instruções. Nas seções seguintes descrevemos os trabalhos anteriores conforme sua classificação.

3.2.1 *Caches* de Dados

Entre os trabalhos que se aplicam somente a *caches* de dados, podemos distinguir os que analisam o código fonte da aplicação e os que particionam o *cache* para armazenar variáveis com propriedades de localidade similares em cada partição de *cache*.

3.2.1.1 Análise Baseada no Código Fonte

Alguns trabalhos determinam a configuração de memória ótima através da análise de código, eliminando a simulação dos rastros de acesso à memória.

Dada uma quantidade de memória, Panda, Dutt e Nicolau [20] analisam o código dos *loops* da aplicação para encontrar o tamanho da memória *scratch-pad*, do *cache* diretamente mapeado e da linha de *cache* que minimizam o número de ciclos de processadores necessários para acessar as matrizes da memória. Ao estimar o desempenho da memória analisando o código dos *loops* da aplicação, esta abordagem dispensa a execução de simulações que demorariam milhares de vezes mais tempo para obter o mesmo resultado. Nesta análise, as referências são agrupadas em classes de equivalência de reuso, onde cada classe é composta por referências com reuso espacial de grupo ou próprio. O reuso espacial de grupo ocorre quando a linha de *cache* acessada por uma referência é reusado por outras [21], como é o caso da referência $A[i][j-1]$ na Figura 3.1, que tem os seus acessos reusados por $A[i][j]$ e $A[i][j+1]$. Já o reuso espacial próprio acontece quando a mesma referência que acessou uma linha de *cache* numa iteração acessa-a novamente na próxima iteração [21], como ocorre com $A[i-1][j]$ e $A[i+1][j]$.

for $i = 1$ to $M - 1$ step 1
  for $j = 1$ to $M - 1$ step 1
    $A[i][j] = A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1] + B[i] + C[j][i]$

Figura 3.1 *Loop* exemplo [20].

Para uma referência $a[i_1][i_2]...[i_m]$ no *loop* de nível m ter sua localidade temporal explorada, a linha de *cache* acessada deve estar presente na próxima iteração. Para isso, a quantidade total de acessos à memória nos *loops* internos ao *loop* de nível m não deve ultrapassar o tamanho do *cache*. Além disso, referências localizadas em linhas diferentes, que aparentemente não exibem localidade espacial de grupo, como $A[i-1][j]$ e $A[i][j]$, podem vir a adquirir localidade, por exemplo, se todos os elementos de uma linha completa da matriz A couberem no *cache*. Deste modo, o dado acessado por $A[i][j]$ em uma iteração pode ser reusado por $A[i-1][j]$ na iteração correspondente a $i + 1$. Como cada conflito de *cache* representa um acesso à memória fora do *chip*, o cálculo do desempenho da memória estima a quantidade de conflitos de *cache*.

O método de projeto proposto em [16] classifica, em tempo de compilação, as estruturas em espaciais, temporais ou não-armazenáveis em *cache*. As estruturas de dados temporais e espaciais são mapeadas na partição temporal e espacial do *cache*, respectivamente, ao passo que as não-armazenáveis em *cache* nunca são trazidas ao *cache*. Nos exemplos utilizados em [16], a partição temporal do *cache* tem até 128 *bytes* e é completamente associativa com linhas de 64 *bits,* para que não seja desperdiçado espaço de linhas maiores com estruturas que apresentam somente localidade temporal. Já a partição espacial tem 8K *bytes* e é diretamente mapeada com linhas de 32 *bytes* para explorar a localidade espacial. Considera-se que uma *word* possui 64 *bits*.

As partições operam independentemente e em paralelo. Mas, em alguns casos, a estrutura pode estar armazenada em ambas as partições. Se uma estrutura *e* é lida da partição temporal e modificada, o nível mais alto de memória não é atualizado se a política de escrita na volta for adotada. Se, mais tarde, uma estrutura próxima de *e* for acessada, e causar uma falta no *cache*, uma cópia antiga de *e* pode ser trazida à partição temporal, caso esta estrutura apresente localidade espacial. Logo, um acerto no *cache* dual ocorre quando a estrutura é encontrada em uma ou ambas as partições. Neste último caso, se o acesso é de leitura, a estrutura é lida da partição temporal. Se o acesso é de escrita, a estrutura é escrita em ambas as partições. A intenção deste particionamento é evitar o desperdício de largura de banda e espaço de memória causado pelo armazenamento em *cache* de estruturas sem nenhuma localidade ou somente com localidade espacial. Deste modo, o deslocamento destas estruturas entre os níveis da hierarquia de memória é minimizado, reduzindo a taxa de faltas.

A classificação das estruturas, em tempo de compilação, é feita através da análise de reuso, volume e interferência das estruturas, sendo aplicada somente às estruturas referenciadas dentro de *loops*. Na análise de reuso, se a mesma estrutura é acessada por uma instrução em mais de uma iteração, a estrutura apresenta reuso temporal. Se, além da estrutura em si, endereços próximos a ela são acessados em mais de uma iteração por uma instrução, a estrutura apresenta reuso espacial. Com base no tipo de reuso da estrutura, limites do *loop*, número de linhas e tamanho de linha no *cache*, o volume de dados acessado entre dois reusos consecutivos é calculado. Se este volume for pequeno o suficiente para permitir que a estrutura seja reusada antes de ser removida do *cache*, a estrutura exibe localidade. Porém, se duas estruturas são mapeadas para posições do *cache* que diferem entre si por um valor menor que a linha de *cache*, elas competem pela mesma linha. Como esta interferência impede que a estrutura permaneça no *cache* a tempo de ser reusada, ela não consegue ter a sua localidade explorada, não sendo armazenada no *cache* dual.

3.2.1.2 Mapeamento de Variáveis em *Caches* Particionados

O mapeamento de variáveis de uma aplicação embarcada em *caches* particionados de acordo com a sua localidade tem sido assunto de diversas publicações.

A técnica apresentada por Lee, Park e Kim [19] adiciona um *buffer* completamente associativo com tamanho de linha grande a um *cache* diretamente mapeado com tamanho de linha pequena e combina-os a uma unidade de *hardware* de *prefetch*. Como o *cache* diretamente mapeado tem a finalidade de explorar a localidade temporal, seu tamanho de linha é projetado para ser o menor possível, de modo a aumentar a quantidade de linhas disponíveis. Para compensar a falta de localidade espacial do *cache* temporal, um *buffer* completamente associativo é configurado com tamanho de linha grande e o *prefetch* de *hardware* busca uma linha grande durante uma falta. Perante uma falta, o bloco requisitado é armazenado primeiramente no *buffer*. Quando algum bloco é retirado do *buffer*, somente os pedaços de bloco que foram acessados no *buffer* durante um determinado intervalo são movidos para o *cache*. Desta forma, esta técnica consegue reduzir de 10 a 60% o consumo de energia.

Assim como o método encontrado em [16], o método proposto em [15] classifica as estruturas em espaciais, temporais ou não-armazenáveis em *cache*. No entanto, em [15], a classificação das estruturas é feita em tempo de execução com o auxílio da tabela de previsão de localidade, que é um *hardware* adicional que armazena informações sobre as instruções de *load* e *store* recentemente executadas, tais como endereço da instrução, último endereço referenciado pela instrução, distância *d* entre o último e penúltimo endereço referenciado pela instrução e número de elementos consecutivos acessados pela instrução nesta distância *d*. A partir destas informações é possível saber se a estrutura acessada é escalar ou matricial, e, no caso da estrutura ser matricial, o tamanho da matriz e a distância entre seus elementos. Considerando que matrizes com distâncias grandes entre seus elementos, variáveis escalares e matrizes com distâncias pequenas entre seus elementos possuem, respectivamente, graus de localidade pequenos, médios e grandes, classifica-se as estruturas em espaciais, temporais ou não-armazenáveis em *cache*.

A metodologia de Naz, Kavi, Rezaei e Li [17], [18] usa *caches* particionados para escalares e matrizes em conjunto com o *cache* de vítimas e *buffer* de *streams*, onde o *stream* consiste de uma coleção de elementos com posições sucessivas, as quais diferem entre si por um valor fixo. As faltas compulsórias são as maiores causadoras de faltas em acessos a matrizes e *streams* porque estas estruturas normalmente possuem somente localidade espacial e não cabem no *cache* devido ao tamanho. Para reduzir estas faltas e explorar a localidade espacial através do *prefetch* de blocos

vizinhos, o *cache* de matrizes é diretamente mapeado com tamanho de linha grande. Em contrapartida, como os escalares apresentam uma taxa alta de faltas por conflito decorrente de sua localidade temporal, para disponibilizar mais linhas, o *cache* de escalares é associativo por conjunto com 2 vias de linhas pequenas. Enquanto o *cache* de vítimas é adicionado à arquitetura por reduzir a taxa de faltas por conflito de dados com localidade temporal, o *buffer* de *streams* é adicionado por diminuir o descarte prematuro de dados com localidade espacial, pois armazena os blocos vizinhos buscados via *prefetch*. O *cache* de vítimas é um *cache* de 4 a 16 linhas completamente associativo, ao passo que o *buffer* de *streams* é um *buffer FIFO* completamente associativo com 4 ou 5 entradas.

Quando ocorre uma falta no *cache*, o *cache* de vítimas é consultado. Caso haja um acerto no *cache* de vítimas, o bloco requisitado vai para o *cache*, enquanto a linha de *cache* que foi desalojada vai para o *cache* de vítimas. Quando há uma falta no *cache* de vítimas, o bloco requisitado e seus blocos sucessivos são buscados do próximo nível de memória. Porém, enquanto o bloco requisitado é armazenado no *cache*, seus blocos sucessivos vão para o *buffer* de *streams*. Como o mapeamento de escalares e matrizes reduz as faltas por conflito e por capacidade, esta metodologia consegue melhorar o tempo médio de acesso, o consumo de energia e o tamanho do *cache*.

3.2.2 *Caches* de Dados e Instruções

Entre os trabalhos que se aplicam tanto a *caches* de dados quanto de instruções, podemos dividir entre os que abordam e os que não abordam *caches* compartilhados por *multicores* (multi-núcleos).

3.2.2.1 *Caches* Compartilhados por *Multicores*

Em trabalhos recentes, as técnicas de projeto da hierarquia de memória para processadores *multicores* utilizam, além do *cache* nível 1, um grande *cache* nível 2, compartilhado entre *multicores*. Com o intuito de prever e aumentar o desempenho destes *caches*, Suhendra e Mitra [5] combinam estratégias de trancamento e particionamento de *cache* para limitar a substituição de blocos e eliminar a interferência entre as tarefas que executam neste grande *cache* nível 2. Assume-se que a coerência de *cache* é implementada em *hardware*.

Quando o *cache* é trancado estaticamente (*statically locked*), seu conteúdo permanece inalterado durante a execução. Quando ele é trancado dinamicamente (*dynamically locked*), seu conteúdo pode ser substituído. Além disso, o *cache* pode ser particionado em tarefas, núcleos, ou simplesmente não ser particionado. Quando o *cache* não é particionado, a linha de *cache* pode ser ocupada por qualquer tarefa de qualquer núcleo. Se os conjuntos do *cache* forem particionados entre as tarefas, os conjuntos de *cache* podem ser ocupados somente pela tarefa para a qual foram atribuídos. Se os conjuntos do *cache* forem particionado entre os núcleos (*cores*), cada conjunto de *cache* pode ser ocupado somente pelo núcleo para o qual foi designado. A análise da combinação destes mecanismos mostra que o particionamento baseado em núcleo apresenta melhores resultados, independente da estratégia de trancamento empregada.

3.2.2.2 *Caches* Não Compartilhados por *Multicores*

Pesquisas recentes, relatadas a seguir, têm mostrado que o desempenho de aplicações embarcadas pode ser melhorado por *caches* ajustados a seus padrões de acesso. Entre as pesquisas que não abordam *caches* compartilhados por *multicores*, podemos distinguir entre as que realizam este ajuste dinamicamente, e as que realizam este ajuste estaticamente.

3.2.2.2.1 Caches Ajustados Dinamicamente a seus Padrões de Acesso

Bornoutian e Orailoglu [12] propõem uma arquitetura capaz de reconfigurar dinamicamente o *cache* para evitar poluição e interferência, reduzindo a taxa de faltas. Para isso, conforme ilustrado na Figura 3.2, duas modificações no *hardware* são realizadas: uma lista circular que armazena os últimos blocos removidos do *cache*, e dois *bits* adicionais em cada conjunto do *cache*, um para expandir o conjunto do *cache* e outro para indicar acertos ocorridos no conjunto expandido.

Diante de uma falta no *cache*, se o bloco requisitado já se encontra na lista circular, há um indício de que pelo menos dois elementos acessados em tempos próximos mapeiam para a mesma linha de *cache*. Esta linha é então dinamicamente expandida habilitando-se o *bit* de expansão para que, numa próxima falta no *cache*, o conjunto secundário seja verificado. Se o dado for encontrado no conjunto secundário, o *bit* que indica acerto é habilitado. Este *bit* é denominado de alternância, pois,

uma vez habilitado, indica que as próximas buscas no *cache* devem iniciar pelo conjunto secundário pois, como o último acerto ocorreu no conjunto secundário, a probabilidade do próximo ocorrer lá é alta. Caso o dado não seja encontrado no conjunto secundário, o *cache* deve buscá-lo no conjunto primário.

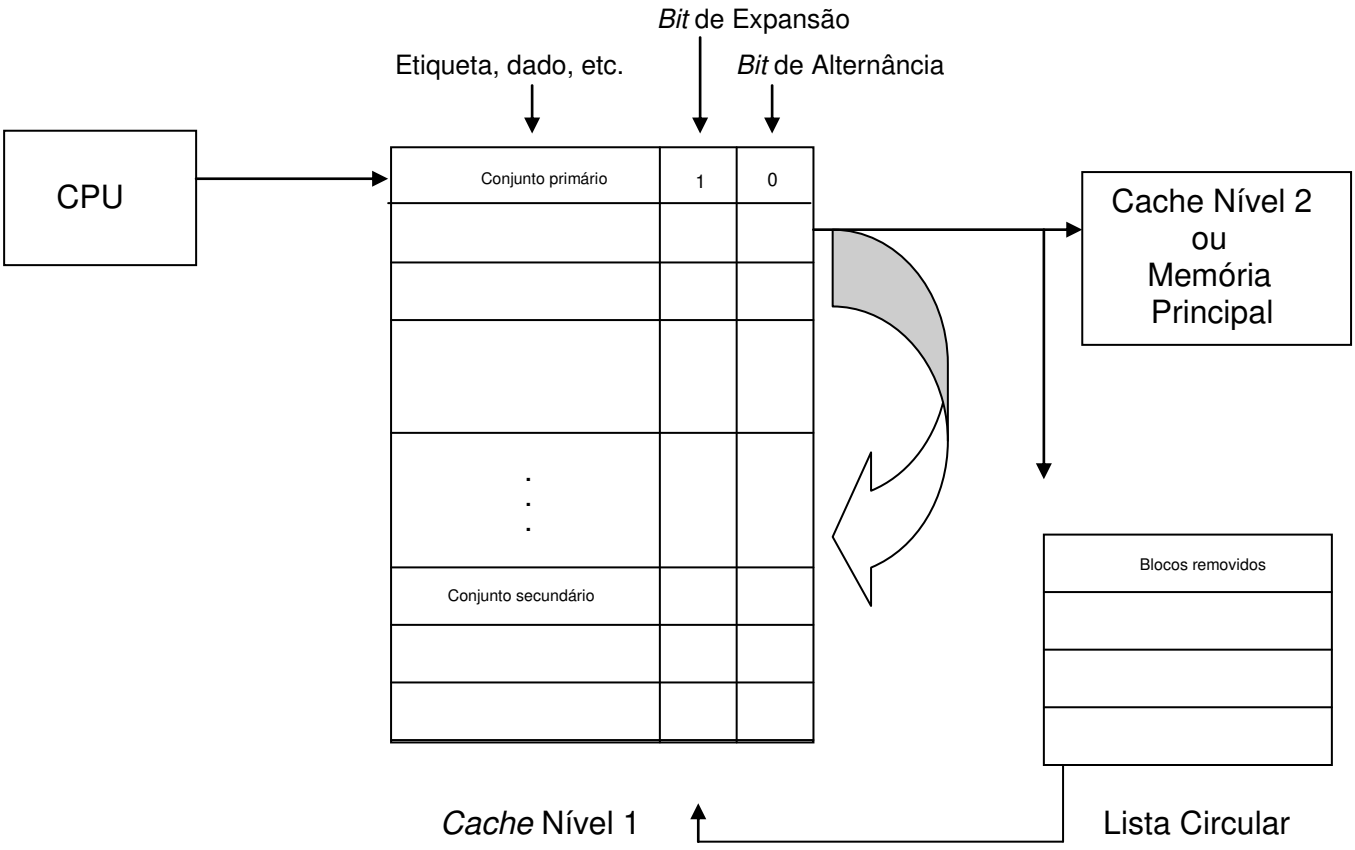


Figura 3.2 Arquitetura de *cache* nível 1 dinamicamente reconfigurável para reduzir taxa de falta [12]

Gordon-Ross, Vahid e Dutt [13] propõem uma heurística para ajustar, em tempo de execução, os parâmetros de *cache* nível 2 unificado reconfigurável levando em conta os parâmetros dos *caches* nível 1 de dados e instruções e o impacto de cada parâmetro na energia total. Neste *cache* nível 2 reconfigurável, as vias podem armazenar somente dados ou instruções, tanto dados quanto instruções ou não serem ativadas. Esta heurística, chamada de Exploração de *Cache* Alternando *Cache* de Instrução, Dados e Unificado com Refinamento de Adição de Vias e descrita na Figura 3.3, é capaz de determinar uma configuração de *cache* que consome em média 61% menos energia explorando somente 0,2% do espaço de projeto.

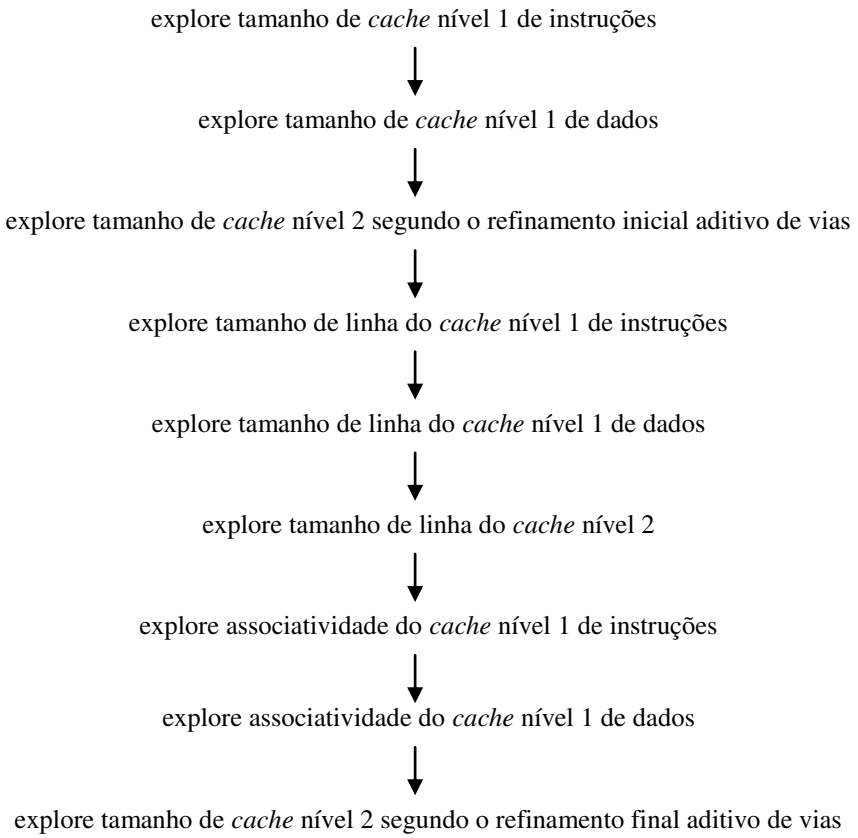


Figura 3.3 Heurística de Exploração de *Cache* Alternando *Cache* de Instrução, Dados e Unificado com Refinamento de Adição de Vias [13].

A heurística passa então a ajustar os tamanhos de linha do *cache* nível 1 de instrução, de dados e do *cache* nível 2 unificado, nesta ordem. Em seguida, são ajustadas as associatividades do *cache* nível 1 de instrução, e depois o de dados. Tanto o ajuste no tamanho de linha quanto de associatividade buscam a configuração com menor consumo de energia.

Por fim, o tamanho de *cache* nível 2 é novamente ajustado na fase de refinamento final aditivo de vias, ilustrado na Figura 3.4 (b). Nesta fase, a heurística busca configurações ainda mais econômicas no consumo de energia através da adição ou remoção de uma via unificada, de dados ou de instruções. Este último ajuste prossegue até que não haja mais configurações inexploradas, ou até que não seja mais possível reduzir o consumo de energia. Como a taxa de faltas do *cache* nível 1 de dados é normalmente maior que a do de instruções, a configuração de *cache* encontrada por esta heurística tende a apresentar mais vias de dados do que de instruções.

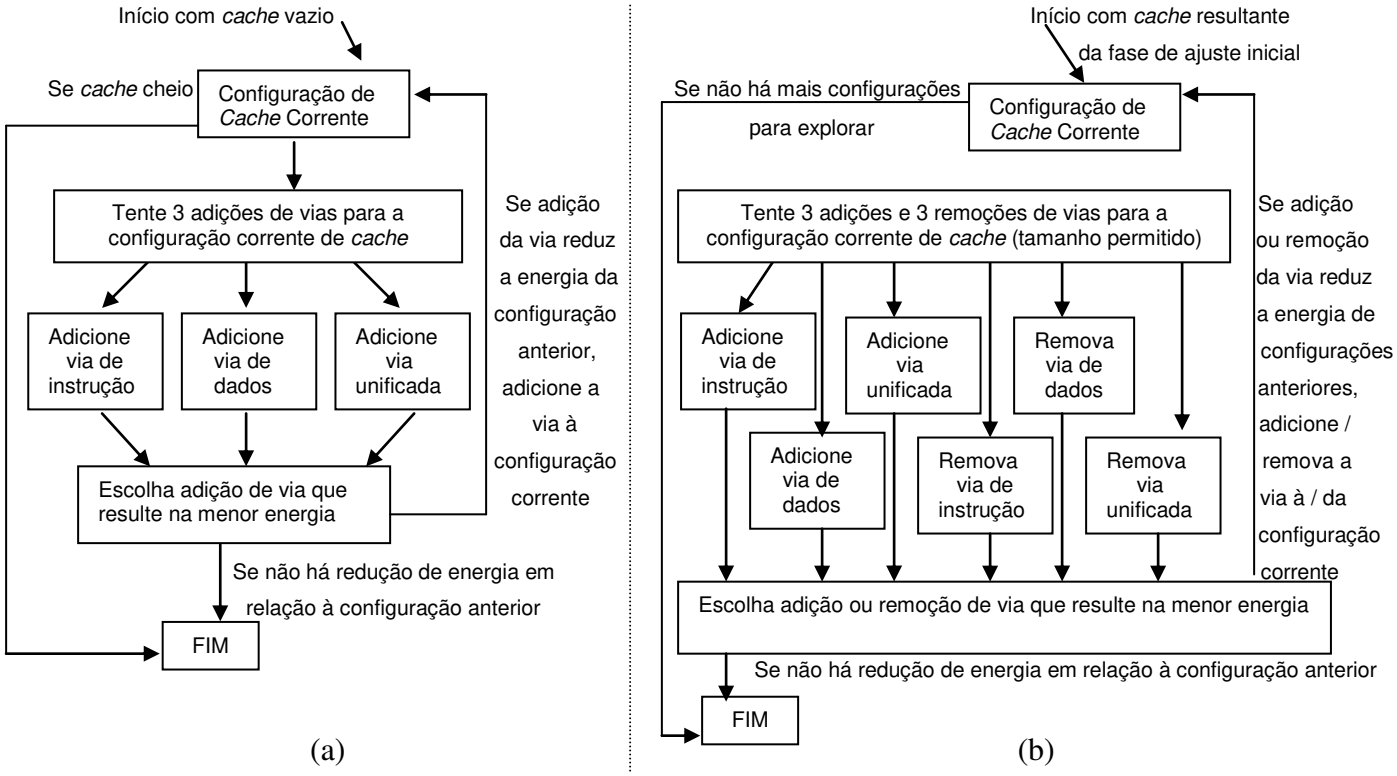


Figura 3.4 Ajuste inicial (a) e adicional (b) de um *cache* nível 2 unificado reconfigurável [13].

Zhang e Vahid [4] investigam o efeito do *cache* de vítimas como parâmetro de memória configurável em *caches* diretamente mapeados com tamanho total, tamanho de linha e associatividade configuráveis. Nesta investigação, o *cache* de vítimas pode ou não ser ativado em tempo de execução. Diante de uma falta no *cache* diretamente mapeado, se o *cache* de vítimas estiver desativado, busca-se o dado direto no próximo nível de memória. Senão, busca-se o dado primeiro no *cache* de vítimas. Caso ele seja encontrado lá, há uma redução no consumo de energia e no tempo de execução pelo fato do *cache* de vítimas estar localizado dentro do *chip*, junto ao *cache* diretamente mapeado. Caso contrário, é necessário buscar o dado no próximo nível de memória, fora do *chip*.

Quando o *cache* de vítimas consegue reduzir o número de acessos à memória fora do *chip* ou reduzir o tamanho total, tamanho de linha e associatividade, sem aumentar o número de acessos ao próximo nível de memória, há redução no consumo de energia e no tempo de execução. Por outro lado, quando nenhuma destas condições é satisfeita, a penalidade da falta aumenta por causa do tempo adicional desperdiçado na consulta ao *cache* de vítimas por um dado que não estava armazenado nele. Porém, como o uso do *cache* de vítimas é configurável, ele pode ser ativado somente em programas de aplicações onde sua taxa de acerto justifique seu uso. Deste modo, conclui-se que o *cache* de vítimas

pode ser incorporado em arquiteturas de sistemas embarcados para permitir uma maior exploração visando atender os requisitos das aplicações na redução do consumo de energia e tempo de execução. Nos exemplos com *caches* diretamente mapeados, há uma redução de até 66% de energia.

3.2.2.2.2 Caches Ajustados Dinamicamente a seus Padrões de Acesso

O método automatizado proposto por Ross, Vahid e Dutt [6] ajusta os parâmetros do *cache* nível 2 para otimizar o consumo de energia de uma arquitetura composta por *caches* níveis 1 e 2, ambos particionados entre dados e instruções. Este método consegue encontrar configurações de *cache* que consomem apenas de 1 a 1,4% mais energia que a configuração ótima explorando somente 6,5% do espaço de projeto. Este resultado é obtido porque os parâmetros de *cache* são explorados segundo a sua ordem de impacto no consumo de energia, conforme descrito por Zhang, Vahid e Lysecky [39]. Este último trabalho busca o parâmetro com maior impacto no consumo de energia, visto que ele corresponde ao parâmetro a ser explorado primeiro. Após manter dois parâmetros fixos e variar o outro, este trabalho mostra que os parâmetros de *cache* com maior impacto no consumo de energia são primeiro o tamanho de *cache*, seguido pelo tamanho de linha de *cache* e, por último, associatividade.

Além disso, em [6], os dois níveis de *cache* são explorados simultaneamente, pois o desempenho de cada nível de *cache* depende do outro. Assim, primeiro busca-se o tamanho de *cache* ótimo para o nível 1 e depois o do nível 2. Em seguida, procura-se o tamanho de linha ótimo para o nível 1 e depois para o nível 2. Por fim, procura-se a associatividade ótima do nível 1 seguida pela do nível 2. Como a determinação de um único valor ótimo de tamanho de *cache* e tamanho de linha de *cache* restringe a exploração de todas as associatividades possíveis, o método considera um último aumento nos tamanhos de *caches* nível 1 e 2 para que todas as associatividades possíveis sejam consideradas. Desta forma, se um aumento no *cache* nível 2 de 16K *bytes* diretamente mapeado para um *cache* nível 2 de 32K *bytes* diretamente mapeado não resultar em melhorias de energia, outras associatividades são exploradas, permitindo buscas num *cache* nível 2 de 32K *bytes* com associatividade igual a 2, por exemplo.

Givargis, Henkel e Vahid [14] exploram os efeitos das interdependências entre os parâmetros de *cache* e parâmetros de barramento de *cache* na área, desempenho e consumo de energia. A partir da simulação dos rastros de acesso à memória, calcula-se o número de acessos, acertos e faltas no *cache.* Com estes dados, calcula-se o número de acessos à memória principal e o número de transferências no

barramento entre *cache* e memória principal, e entre *cache* e CPU, de modo que a energia consumida por cada parte do sistema embarcado é calculada. Analisando estes cálculos observa-se que uma mesma aplicação sob diferentes configurações de parâmetros de *cache* (tamanho de *cache,* tamanho de linha, associatividade) e de interfaces (largura de barramento, codificação de barramento) pode ser responsável pelo consumo de 3 a 41% da razão da energia consumida pelo barramento pela energia consumida por todos os *cores* envolvidos, como *cache*, memória principal, CPU, etc. Deste modo, conclui-se que as melhores configurações de *caches* e barramento são interdependentes.

Este trabalho [14] também conclui que, ao contrário de tecnologias mais antigas, como a de 0,80µ, tecnologias mais recentes, como a de 0,18µ, apresentam um consumo de energia de barramento inversamente proporcional ao tempo de execução. A explicação fornecida é que tecnologias mais recentes possuem menos fios que tecnologias mais antigas, e, menos fios gastam menos energia, mas precisam de mais transferências entre barramentos, aumentando o tempo de execução. Por fim, estes cálculos mostram que o maior responsável pelo consumo de energia em tecnologias mais recentes é o barramento entre CPU e *caches*. Por isso, para a seleção de um sistema com área, desempenho e consumo de energia otimizados, Givargis, Henkel e Vahid [14] sugerem primeiro a determinação do tamanho do barramento entre CPU e *caches* seguido do cálculo da configuração de *cache* e barramento.

Assumindo um tamanho fixo de linha de *cache*, Ghosh e Givargis [40] encontram os pares de associatividade e número de linhas de *cache* que satisfazem a restrição de número de faltas analisando os endereços das referências à memória, pois a partir destes endereços é possível saber quais referências são mapeadas para cada conjunto de linha. Como as faltas compulsórias não podem ser evitadas sem alterar o tamanho de linha de *cache*, elas não são consideradas na restrição de número de faltas. As referências acessadas entre dois acessos consecutivos a uma referência *Ref* compõem o conjunto de referências potencialmente conflitantes com *Ref*. Para o levantamento dos conjuntos conflitantes, o primeiro acesso a *Ref* é descartado por constituir a falta compulsória. A falta no *cache* é computada quando a intersecção entre o conjunto de referências mapeadas para a mesma linha de *Ref* e o conjunto potencialmente conflitante de *Ref* tem cardinalidade maior que a associatividade sendo avaliada. A associatividade é então incrementada até que a restrição no número de faltas seja alcançada.

3.3 Conclusão

Neste capítulo comentamos alguns algoritmos desenvolvidos em trabalhos anteriores para otimizar a hierarquia de memória em termos de energia e desempenho.

Abordamos algoritmos que se aplicam somente a *caches* de dados e algoritmos que se aplicam tanto a *caches* de dados quanto de instruções. Entre os algoritmos que se aplicam somente a *caches* de dados, descrevemos algoritmos que analisam o código fonte da aplicação e algoritmos que mapeiam as variáveis em partições distintas do *cache*. Entre os algoritmos que se aplicam tanto a *caches* de dados quanto de instruções, distinguimos entre os que utilizam e os que não utilizam *caches multicores*. Dentre estes últimos, comentamos algoritmos que encontram a melhor configuração de memória dinâmica e estaticamente.

Podemos notar que o *cache* de vítimas, *buffer* de *streams* e *hardware* para *prefetch* podem ser empregados na configuração de memória para melhorar os resultados. Observamos também que o ajuste dos parâmetros do *cache* depende dos parâmetros do outro nível de *cache* e da configuração do barramento. Analisamos também trabalhos que promovem alterações no *hardware* convencional do *cache*.

No próximo capítulo detalharemos nosso algoritmo de ajuste da configuração da hierarquia de memória através do mapeamento de matrizes em *caches* particionados.

Capítulo 4

Descrição do Algoritmo

4.1 Introdução

Neste capítulo, descrevemos nossa metodologia para projetar *caches* de matrizes de duas partições com uma restrição de tamanho. Um *cache* de matrizes é um *cache* de dados para vetores, matrizes e outras estruturas de dados complexas de uma aplicação embarcada. Por simplicidade, nos referimos a todas estas estruturas como matrizes. Um *cache* de matrizes particionado é definido pela organização de suas partições e pelo mapeamento das matrizes em suas partições.

Não foi considerado neste trabalho armazenamento em *cache* de escalares. Existem trabalhos para dimensionamento de *caches* para escalares [17], [18] e para armazenamento de escalares em *scratch-pads* [43].

Nossa metodologia procura por *caches* de matrizes de duas partições que minimizam o tempo médio de acesso à memória, conforme definido pela equação 2.7 no Capítulo 2. Os *caches* de matrizes de duas partições considerados na busca são gerados a partir de *caches* associativos por conjunto que satisfazem a restrição de tamanho.

O projeto começa com um *cache* de matrizes associativo por conjunto e usa simulação baseada em rastros de acesso à memória para introduzir uma métrica, a ser definida neste capítulo, que quantifica a alternância de acessos às matrizes entre metades das linhas de *cache*. As matrizes com menores alternâncias de acesso são mapeadas para uma nova partição de *cache* com metade do tamanho de linha, que é gerado através da reconfiguração de algumas vias do *cache* original. Esta

reconfiguração ocorre durante a fase de projeto de *caches* particionados, podendo também ser executado durante a compilação , através de compiladores especializados.

Este procedimento é repetido para vários *caches* de matrizes unificados associativos por conjunto e os *caches* de matrizes particionados correspondentes ao menor tempo médio de acesso à memória são selecionados.

4.2 Etapas do Algoritmo

Nosso algoritmo é composto por 6 etapas:

- *Instrumentação da Aplicação*;
- *Simulação da Aplicação*;
- *Simulação de Cache I*;
- *Particionamento de Cache*;
- *Simulação de Cache II*;
- *Avaliação dos Caches*.

Durante a etapa de *Instrumentação da Aplicação*, todos os acessos à memória correspondentes à matrizes são instrumentados.

No etapa de *Simulação de Aplicação*, os rastros de acesso à memória correspondentes às matrizes são gerados através da simulação de uma versão do código da aplicação embarcada com acessos a matrizes instrumentados. Entradas típicas devem ser utilizadas para esta simulação.

Na etapa de Simulação de *Cache I*, o espaço de *cache* disponível, de *c words* de dados, é organizado como um *cache* C_0 associativo de matrizes com *n* vias e linhas de *b words*. Para cada valor considerado de *b* e *n*, uma simulação baseada nos rastros de acesso à memória mapeia os acessos feitos às matrizes às linhas de *cache*. Para cada matriz, é computada uma métrica *M* que quantifica a fração de acessos que é mapeada numa metade distinta da linha em relação ao acesso anterior feito nesta mesma linha. A simulação também estima a taxa de faltas de C_0.

Na etapa de Particionamento de *Cache* , as matrizes são ordenadas de acordo com os valores de *M* e um limite *MLIM* é definido, conforme cálculos a serem mostrados neste Capítulo, de modo que as matrizes com valores maiores que *MLIM* são mapeadas para o *cache* associativo C_1 de n_1 vias e tamanho de linha *b*, enquanto as outras matrizes são mapeadas para o *cache* associativo C_2 de n_2 vias e tamanho de linha *b*/2. Os graus de associatividade n_1 e n_2 são calculados a partir do número de acessos

distintos a blocos correspondendo às matrizes mapeadas para C_1 e C_2, com $n_1 + n_2 = n$, de forma que o tamanho total do *cache* particionado permaneça igual à c. Os *caches* C_0, C_1 e C_2 substituem seus blocos segundo a estratégia do bloco menos recentemente usado (LRU). A Figura 4.1 ilustra um exemplo de particionamento de *caches,* onde o cache C_0 com n vias de m linhas de tamanho b é particionado no *cache* C_1 com n-1 vias de m linhas de tamanho b e no *cache* C_2 com 1 via de 2m linhas de tamanho b/2. As linhas de um mesmo conjunto estão sombreados.

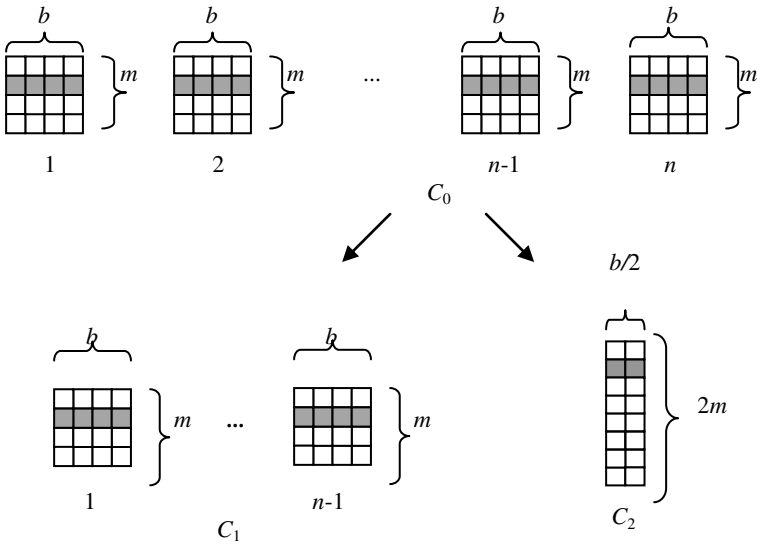


Figura 4.1 Exemplo de mapeamento de endereço da instrução à partição de *cache*.

Novas simulações são executadas para obter as taxas de faltas de C_1 e C_2 durante a etapa de *Simulação de Cache II*.

Na etapa final, as informações obtidas nas simulações são utilizadas para calcular o tempo médio de acesso à memória e o consumo de energia dos *caches* particionados e unificados. Os *caches* de matrizes particionados com menor tempo médio de acesso à memória são selecionados para serem comparados com o *cache* de matrizes unificado de menor tempo médio de acesso à memória.

O projeto de *caches* nível 1 envolve um compromisso entre a taxa de falta, o tempo de acerto e a penalidade da falta. Em nossa metodologia, o particionamento de *cache* reduz o tamanho e a associatividade do *cache*, além do tamanho de linha, para a partição C_2. Como o tempo de acerto geralmente aumenta com o tamanho do *cache* e associatividade, cada partição dos *caches* de matrizes particionados possuem um tempo de acerto menor que o do *cache* de matrizes unificado. Por outro lado, o particionamento de *caches* pode aumentar severamente a taxa de falta. Neste trabalho, matrizes

com valores próximos de *M* são mapeadas na mesma partição de *cache*, a qual tem uma customização configurada para estas matrizes como forma de reduzir a taxa de falta.

As seções 4.2.1 a 4.2.6 descrevem detalhadamente as etapas da nossa metodologia de projeto. A seção 4.2.7 explica como esta metodologia pode ser aplicada iterativamente para gerar *caches* de matrizes particionados com mais de duas partições.

4.2.1 Fase de *Instrumentação da Aplicação*

Para a geração dos rastros de acesso à memória, o código fonte é manualmente instrumentado para escrever num arquivo de saída os acessos feitos à memória. Consideramos todos os acessos feitos às matrizes e estruturas, sendo excluídas somente as variáveis escalares.

Escrevemos numa linha do arquivo de saída o nome da matriz, endereço da matriz e tamanho do dado acessado em *bytes*, seguindo o padrão *<nome da matriz>;<endereço>;<tamanho em bytes>*. No exemplo da Figura 4.2, o endereço *0x4FF8F0* da matriz *ip* contendo um dado de *4 bytes* é acessado. Em seguida, o endereço *0x004BF8B0* da matriz *a* contendo um dado de *8 bytes* é acessado.

```
ip;0x4FF8F0;4;
a;0x4BF8B0;8;
a;0x4BF8B4;8;
a;0x4BF8B0;8;
a;0x4BF8B4;8;
```

Figura 4.2 Exemplo de rastro de acesso à memória.

Consideramos alocações globais, locais, estáticas e dinâmicas. Como o algoritmo assume que um mesmo endereço é usado por somente uma matriz, e como as alocações locais podem reservar espaços de memória diferentes em cada alocação, as variáveis locais das aplicações foram transformadas em estáticas. Estas variáveis estáticas têm escopo limitado ao seu módulo, mas permanecem armazenadas ao longo da execução do programa [35].

Matrizes acessadas nas funções da biblioteca *C* e *C++* chamadas pelas aplicações não foram instrumentadas. Assim, a função *fopen*, por exemplo, que é chamada de dentro do aplicação JPEG, não foi instrumentada. Os trechos de código onde a matriz era declarada, alocada ou desalocada não foram considerados na instrumentação por não gerarem acessos à memória, mas somente reservarem um espaço de memória.

No caso de parâmetros passados por referência e ponteiros referenciando uma memória anteriormente alocada que podem ser associados a várias matrizes, a matriz efetivamente acessada no momento da execução é identificada verificando-se a qual dos intervalos de endereço de matrizes este endereço pertence.

Nosso algoritmo considera que cada acesso registrado no rastro gera um acesso ao *cache* de matrizes, e nosso simulador de *cache* assume que os acessos ao *cache* são feitos em *words*. Tendo em vista que o tamanho do tipo *word* armazenado pela matriz pode variar de acordo com o sistema operacional, calculamos este tamanho pelo número de *bytes* registrado no rastro. Nas simulações foram utilizados os tamanhos em *bytes* mostrados na Tabela 4.1.

Tabela 4.1 Tamanho em *bytes* dos tipos.

Tipo	Tamanho (*bytes*)
char	1
unsigned char	1
short	2
int	4
unsigned int	4
word	4
float	4
double	8

Como o tipo *double* contém 8 *bytes*, que é exatamente o dobro do tamanho da *word*, o rastro de acesso a uma variável do tipo *double* no endereço *e* inclui dois acessos no rastro: um acesso à posição *e* e outro à posição (*e* + 4), correspondente a segunda *word*. É por isso que no rastro da Figura 4.2 o acesso ao endereço 0x4BF8B0 da matriz *a* contendo um dado de *8 bytes* é seguido do acesso ao endereço 0x4BF8B4 da matriz *a*.

No caso de acessos a tipos com tamanhos menores que uma *word*, verificamos que o compilador não insere espaços para igualar ao tamanho de uma *word*, mantendo seu tamanho original. Por isso, assumimos que estes tipos são compactados. Por exemplo, 4 caracteres de 8 *bits* são agrupados em uma *word* de 32 *bits*.

Para observarmos os padrões de acesso de cada matriz individualmente, cada linha de *cache* deve armazenar dados de somente uma matriz por vez. Para isso, inserimos espaços nas matrizes que não possuem tamanhos múltiplos do tamanho da linha de *cache*, garantindo que nenhuma outra matriz ocupará simultaneamente a porção de linha deixada desocupada. É uma prática comum inserir espaços

para evitar faltas por conflito. Como a quantidade de espaços inseridos depende do tamanho de linha, inserimos espaços referentes ao maior tamanho de linha simulado, pois ele é sempre múltiplo dos outros tamanhos de linha. Desta forma, os espaços não precisam ser inseridos para cada tamanho de linha simulado.

A quantidade de memória adicionada por esta inserção de espaços é calculada pela Equação 4.1.

$$Espaço\_Adicional = \sum_i [b - (tamanho(A_i) MOD(b))] MOD(b) \quad \text{(eq. 4.1)}$$

Para cada matriz A_i, a expressão ($tamanho(A_i)$ *MOD* (b)) retorna o número de *words* da matriz A_i que não cabe numa linha inteira do *cache*, implicando na inserção de [b - ($tamanho(A_i)$ *MOD* (b))] *MOD* (b) espaços para o preenchimento do restante do tamanho da linha. O segundo *MOD* (b) é inserido na equação para que o espaço adicional não seja igual à b quando a expressão ($tamanho(A_i)$ *MOD* (b)) é igual à zero. Quando esta expressão é diferente de zero, a inclusão da segunda operação *MOD* (b) não altera o resultado.

4.2.2 Fase de *Simulação da Aplicação*

Nesta fase, o código instrumentado na fase anterior é executado para um conjunto de entradas típicas e como resultado temos um rastro com $NREF_0$ *words* referenciadas correspondentes às matrizes da aplicação.

4.2.3 Fase de *Simulação de Cache I*

Nesta etapa ocorre a simulação do *cache* associativo de matrizes unificado C_0 com tamanho de linha b, associatividade n, tamanho total c e política de substituição do bloco menos recentemente usado (*LRU*) no conjunto.

Durante a simulação, diversas variáveis são usadas para acompanhar os acessos no *cache*. O simulador desenvolvido neste trabalho armazena:

- o número total N_t de blocos distintos acessados;
- o número de acertos $NA(C_0)$;
- o número de faltas $NF(C_0)$;

- a via *v* atualmente consultada, que pode assumir valores de 1 a *n*;
- a etiqueta *Etiqueta*(*L*), para cada linha *L*;
- a última metade *Última*(*L*) de linha acessada, para cada linha *L*;
- o número de alternâncias *Alternâncias*(*L*) nos acessos às metades da linha, para cada linha *L* do *cache*;
- o número de acessos *Acessos*(*L*) à linha, para cada linha *L* do *cache*;
- a classificação do último bloco acessado como acerto ou falta em *acertoCache*.

Além disso, as seguintes variáveis são usadas para acompanhar os acessos aos blocos das matrizes:

- o número de alternâncias *Alternâncias*(*B, A*) nos acessos às metades do bloco, para cada bloco *B* de uma matriz *A*;
- o número de acessos *Acessos*(*B, A*) ao bloco, para cada bloco *B* de uma matriz *A*.

Detalhes do nosso simulador de *cache* são descritos no Apêndice I.

A Figura 4.5 mostra o pseudocódigo da etapa de *Simulação de Cache I*.

Para cada referência *Ref* do rastro de acessos à memória, o simulador determina o conjunto de linha de *cache* *Conjunto*(*Ref*) e a etiqueta *Etiqueta*(*Ref*), nas linhas 10 e 11, respectivamente. Em seguida, ele busca, na linha 15, a *Etiqueta*(*Ref*) em alguma das *n* vias *v* deste *Conjunto*(*Ref*), pois cada linha de *cache* é identificada unicamente pelo seu par (*Conjunto, v*).

Se a etiqueta correspondente à referência *Ref*, *Etiqueta*(*Ref*), for a mesma que a de uma linha *L*, temos um acerto no *cache*, na linha 16, e $NA(C_0)$ é incrementado, na linha 17. Se a metade de linha acessada for a mesma que *Última*(*L*), o número de *Alternâncias*(*L*) permanece com o mesmo valor. Se a metade de linha acessada for diferente de *Última*(*L*), o número de *Alternâncias*(*L*) nos acessos às metades da linha é incrementado, na linha 20, e esta nova metade de linha acessada é registrada como *Última*(*L*), na linha 21.

Independente da metade de linha acessada for ou não a mesma que *Última*(*L*), o número de *Acessos*(*L*) é sempre incrementado na linha 18. Se a etiqueta não for encontrada em nenhuma das vias, temos uma falta no *cache*. O número de faltas $NF(C_0)$ é incrementado na linha 24.

Toda vez que ocorre uma falta no *cache*, o algoritmo adiciona, nas linhas 25 e 26, os valores de *Alternâncias*(*L*) e *Acessos*(*L*) aos valores que se encontram em *Alternâncias*(*B, A*) e *Acessos*(*B, A*). Desta forma, os valores das *Alternâncias*(*L*) e *Acessos*(*L*) do bloco *B* que se encontrava no *cache* e que é referente à matriz *A* são atualizados. Como houve uma falta no *cache*, a etiqueta da referência

Etiqueta(*Ref*) substitui *Etiqueta*(*L'*) do bloco menos recentemente usado *L'* neste conjunto de bloco na linha 28. Uma vez que um novo bloco de memória é trazido ao *cache*, os valores de *Alternâncias*(*L*), *Acessos*(*L*) e *Última*(*L*) são inicializados na linhas 29 a 31.

Depois que todas as referências *Ref* do rastro de acesso à memória são consideradas, o simulador copia, nas linhas 35 e 36, o número de *Alternâncias*(*L*) e *Acessos*(*L*) de todos os blocos *B* de cada matriz *A* ainda presentes no *cache* para *Alternâncias*(*B, A*) e *Acessos*(*B, A*), respectivamente. Como este procedimento só é feito no momento de substituição do bloco de *cache*, precisamos fazer esta última cópia quando todos os acessos ao *cache* terminam.

O algoritmo passa então a calcular, na linha 38, os valores de *M*(*B, A*) para todos os blocos *B* de todas as matrizes *A* para em seguida calcular na linha 40 os valores de *M*(*A*) para todas as matrizes *A*.

M(*B, A*) consiste na fração dos acessos à *B* que provocam alternâncias nas metades da linha acessada e é calculada pela Equação 4.2 dividindo o valor de *Alternâncias*(*B, A*) por *Acessos*(*B, A*). Deste modo, esta fração quantifica o número de vezes que os acessos à *B* de uma matriz *A* não foram feitos na mesma metade de linha que o último acesso a esta linha, armazenado em *Última*(*L*).

$$M(B,A) = \frac{Alternâncias(B,A)}{Acessos(B,A)} \qquad \text{(eq. 4.2)}$$

De certa forma, a fração de alternância *M*(*B, A*) avalia se o tamanho do linha é adequado à matriz. Quanto maior o valor de *M*(*B, A*), maior a quantidade de acessos sucessivos à *B* da matriz *A* que exploram suas duas metades, constituindo um indício de que o tamanho da linha do *cache* está bem ajustado à aplicação. Por outro lado, valores baixos de *M*(*B, A*) sinalizam que os acessos a linha se concentram em somente uma de suas metades. Logo, blocos com valores baixos de *M*(*B, A*) são candidatos a terem seus tamanhos ajustados para *b*/2 para liberar sua metade de bloco pouco acessada para outros acessos.

Nas Figuras 4.4(a) e 4.4(b), por exemplo, ilustramos dois casos distintos de padrões de acesso as linhas de *cache* de 8 posições. As posições acessadas estão sombreadas. Na Figura 4.4(a), todos os acessos, de 1 a 4, foram feitos na segunda metade da linha. Logo, sabemos que existem pelo menos quatro posições no início da linha que não foram referenciadas e que, portanto, podem ser liberadas. Na Figura 4.4(b) vemos que do acesso 1 ao 2, a metade de linha acessada passa da segunda metade para a primeira. Dos acessos 2 ao 3 e do 3 ao 4, há novas alternâncias no acesso às metades da linha. Estas3 alternâncias nas metades de linha acessadas mostram uma melhor utilização da linha.

```
1 Simule_Cache (c, b, n, rastro_de_acessos_à_memória)
2   N_t = 0;
3   NA(C_0) = 0;
4   NF(C_0) = 0;

5   para cada linha L do cache
6       Acessos(L) = 0;
7       Alternâncias(L) = 0;
8       Última(L) = Primeira Metade da Linha;

9   enquanto houver Ref não lida no rastro_de_acessos_à_memória
10      calcule Conjunto(Ref);
11      calcule Etiqueta(Ref);
12      acertoCache = falso;
13      v = 0;

14      enquanto ( (!acertoCache) && (v < n))
15          se (Etiqueta(L(Conjunto(Ref), v)) == Etiqueta(Ref))
16              acertoCache = verdadeiro;
17              NA(C_0)++;
18              Acessos(L)++;

19              se (Última(L) mudou)
20                  Alternâncias(L)++;
21                  atualize Última(L);
22          v++;

23      se (!acertoCache)
24          NF(C_0)++;
25          Alternâncias(B, A) += Alternâncias(L);
26          Acessos(B, A) += Acessos(L);
27          encontre a linha L' menos recentemente usado de Conjunto(Ref);
28          substitua L' por L;
29          Alternâncias(L) = 0;
30          Acessos(L) = 1;
31          atualize Última(L);

32      se (linha distinta acessada)
33          N_t++;

34  para todas as linhas L remanescentes no cache
35      Alternâncias(B, A) += Alternâncias(L);
36      Acessos(B, A) += Acessos(L);

37  para todos os blocos B de todas as matrizes A
38      calcule M(B, A);
39  para todas as matrizes A
40      calcule M(A);
41  Fim de Simule_Cache
```

Figura 4.3 Procedimento da fase de *Simulação de Cache I.*

Para cada matriz *A* acessada, o algoritmo calcula pela equação 4.3 a fração de alternância *M*(*A*) da matriz *A* como sendo a mediana das frações de alternância *M*(*B, A*) dos seus blocos correspondentes.

$$M(A) = \text{Mediana de } M(B, A) \text{ para todos blocos } B \text{ de } A \text{ acessados} \qquad \text{(eq. 4.3)}$$

A mediana de *M*(*B, A*) é usada para calcular *M*(*A*) por representar o valor de *M*(*B, A*) da maior parte dos blocos de A.

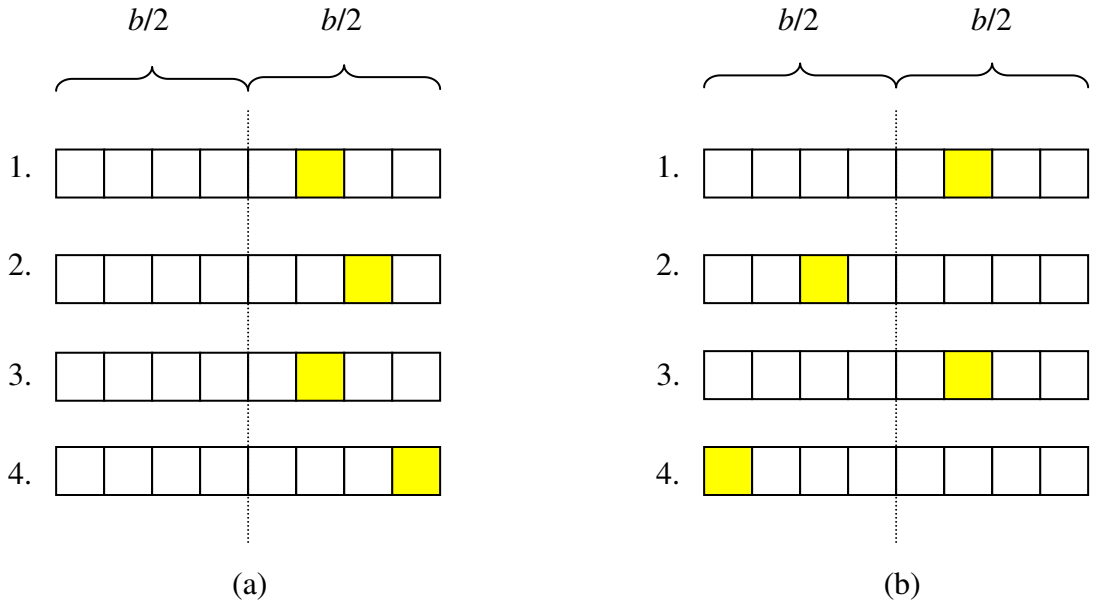


Figura 4.4 Exemplos de valores de *M*(*B, A*) baixo (a) e alto (b).

No exemplo da Figura 4.5, todos os valores de *M*(*B, A*) correspondentes à matriz *A* são ordenados em forma crescente. O valor de *M*(*B, A*) que correspondente à mediana de todos os valores de *M*(*B, A*) é 0,60. Logo, este valor é considerado o fator de alternância *M*(*A*) da matriz *A*.

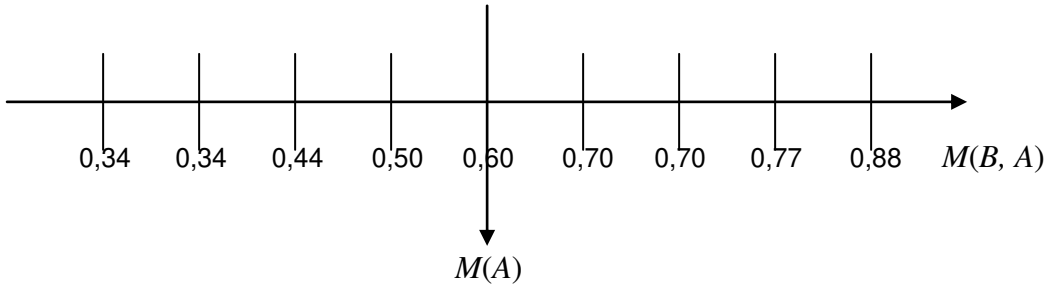


Figura 4.5 Exemplo de cálculo do *M*(*A*).

Apesar de um bloco de memória poder entrar e sair diversas vezes do *cache*, contabilizamos a somatória de todas as *Alternâncias*(L) e *Acessos*(L) correspondentes a cada vez que o bloco permanece no *cache* que ocorrem ao longo da execução completa da aplicação.

Os resultados desta fase do algoritmo que são usados nas fases posteriores são:

- o número $NF(C_0)$ de faltas no *cache* de matrizes unificado;
- o número total $N_t(C_0)$ de blocos distintos referenciados no *cache* de matrizes unificado;
- o número $N(A, C_0)$ de blocos distintos referenciados de cada matriz A;
- a fração $M(A)$ dos acessos de cada matriz A que causam alternâncias nos acessos às metades do bloco da matriz.

4.2.4 Fase de *Particionamento de Cache*

O pseudocódigo desta fase do algoritmo é ilustrado na Figura 4.6.

Para facilitar a implementação de *hardware* de *caches* particionados, nosso algoritmo supõe que uma mesma matriz A não pode ter parte da sua memória mapeada em um *cache* e parte em outro. Por isso, o valor de $M(A)$ é calculado como sendo a mediana dos valores de $M(B, A)$ de todos os blocos de A acessados.

A fase de *Particionamento de Cache* começa na linha 2, onde todos os valores de $M(A)$ encontrados são ordenados em ordem crescente. Para cada valor distinto de $M(A)$, a fração de alternância limite *MLIM* assume o valor de um $M(A)$ na linha 5, de modo que na linha 8 matrizes com valores maiores que *MLIM* são mapeadas para um *cache* C_1 associativo de n_1 vias com tamanho de linha b, enquanto na linha 10 as outras matrizes são mapeadas para um *cache* C_2 associativo de n_2 vias com tamanho de linha $b/2$.

Como a fração de alternância $M(A)$ da matriz é a mediana das frações de alternância $M(B,A)$ dos seus blocos correspondentes, escolhemos a partição de armazenamento da matriz considerando a maioria dos padrões de acesso dos seus blocos acessados.

Como nosso algoritmo busca a melhor solução no espaço de projeto, esta operação é repetida para todos os valores de *MLIM* possíveis no *loop* da linha 4.

Para um determinado valor de *MLIM*, os graus de associatividade n_1 e n_2 são calculados a partir do número $N(A, C_0)$ de acessos distintos aos blocos das matrizes A mapeadas para os *caches* particionados, como mostrado nas Equações 4.4 e 4.5, a seguir.

$$n_1 = \left\lfloor \left(n \times \sum_{\substack{A\ mapeadas \\ para C_1}} N(A, C_0) / N_t(C_0) \right) + 0{,}5 \right\rfloor \qquad \text{(eq. 4.4)}$$

$$n_2 = n - n_1 \qquad \text{(eq. 4.5)}$$

```
1  Particione_Cache(b, n, m, Mi(A) para todas as matrizes A);
2  ordene em ordem crescente os valores de M(A) encontrados para todas as matrizes A;
3  MLIM = 0;
4  para cada valor distinto de M(A)
5      MLIM = M(A);
6      para cada matriz A acessada
7          se (M(A) > MLIM)
8              mapeie A para C1;
9          senão
10             mapeie A para C2;
11     calcule n1 e n2;
12     se ( (n1 > 0) e (n1 < n) )
13         configure C1 como (b, n1, m);
14         configure C2 como (b/2, n2, 2m);
15         selecione (C1|C2) para fase de Simulação de Cache II;
16     se (n1 = 0)
17         sai do loop;
18 Fim de Particione_Cache
```

Figura 4.6 Procedimento da fase de *Particionamento de Caches*.

Como a associatividade é um valor inteiro, na Equação 4.4, o valor 0,5 é somado para arredondar para o inteiro mais próximo o valor entre parênteses. Se n_1 for igual à 0, todas as matrizes de C_0 passam a ser mapeadas em um *cache* de n vias de linhas de tamanho $b/2$. Se n_2 for igual à 0, a etapa de particionamento de *caches* não é executada para seu respectivo valor de *MLIM*. Além disso, como avaliamos *MLIM* em ordem crescente, quanto maior o valor de *MLIM*, maior a quantidade de matrizes mapeadas para C_2.

A Equação 4.5 é decorrente da condição dos *caches* de matrizes particionados terem, juntos, o mesmo tamanho que o *cache* de matrizes unificado C_0. Se C_0 tem m conjuntos de linhas de tamanho b, como $c = n \times m \times b$, e como n satisfaz a Equação 4.5, o número de conjuntos de linhas de C_1 e C_2 é $(m \times b)/b = m$, e $(m \times b/2)/b = m/2$, respectivamente, como podemos ver pela Figura 4.7.

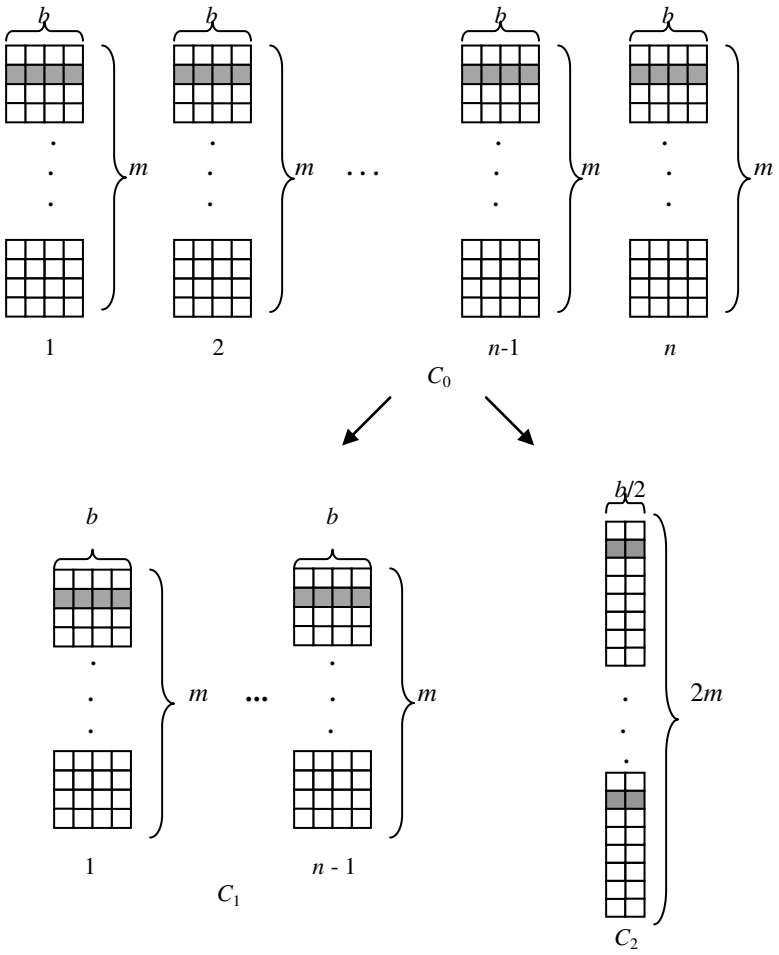


Figura 4.7 Exemplo de configuração de *caches* particionados.

Observe que o número $N(A, C_0)$ de blocos distintos acessados, e não o número total de acessos aos blocos das matrizes A mapeadas à C_1, é utilizado na Equação 4.4, para refletir o uso efetivo do *cache* pela matriz A.

Para cada *MLIM*, um *cache* particionado $(C_1|C_2)$ e mapeamento de matrizes nas partições são especificados. No entanto, como vemos na Figura 4.8, um mesmo *cache* particionado $(C_1|C_2)$ pode ter mais de um mapeamento de matrizes. Se a matriz A acessar uma quantidade tão pequena de blocos distintos que sua adição à C_1 não altere o valor de n_1, os valores de n_1 e n_2 são os mesmos para

diferentes valores de *MLIM*. Devido à ordenação de matrizes por alternância de acessos, o número de *caches* de matrizes particionados baseados em rastros de acesso à memória gerados do *cache* de matrizes unificado é sempre menor que o número de matrizes.

Optamos por particionar o *cache* C_0 fixando o tamanho de linha de C_1 e C_2 e calculando o valor da associatividade através da Equação 4.4 porque a associatividade de C_1 e C_2 pode assumir qualquer valor inteiro no intervalo de 1 a n, ao passo que o tamanho de linha precisa ser uma potência de dois. Desta forma, um *cache* de matrizes unificado C_0 de n vias, pode ser decomposto em até (n-1) *cache* particionados ($C_1|C_2$).

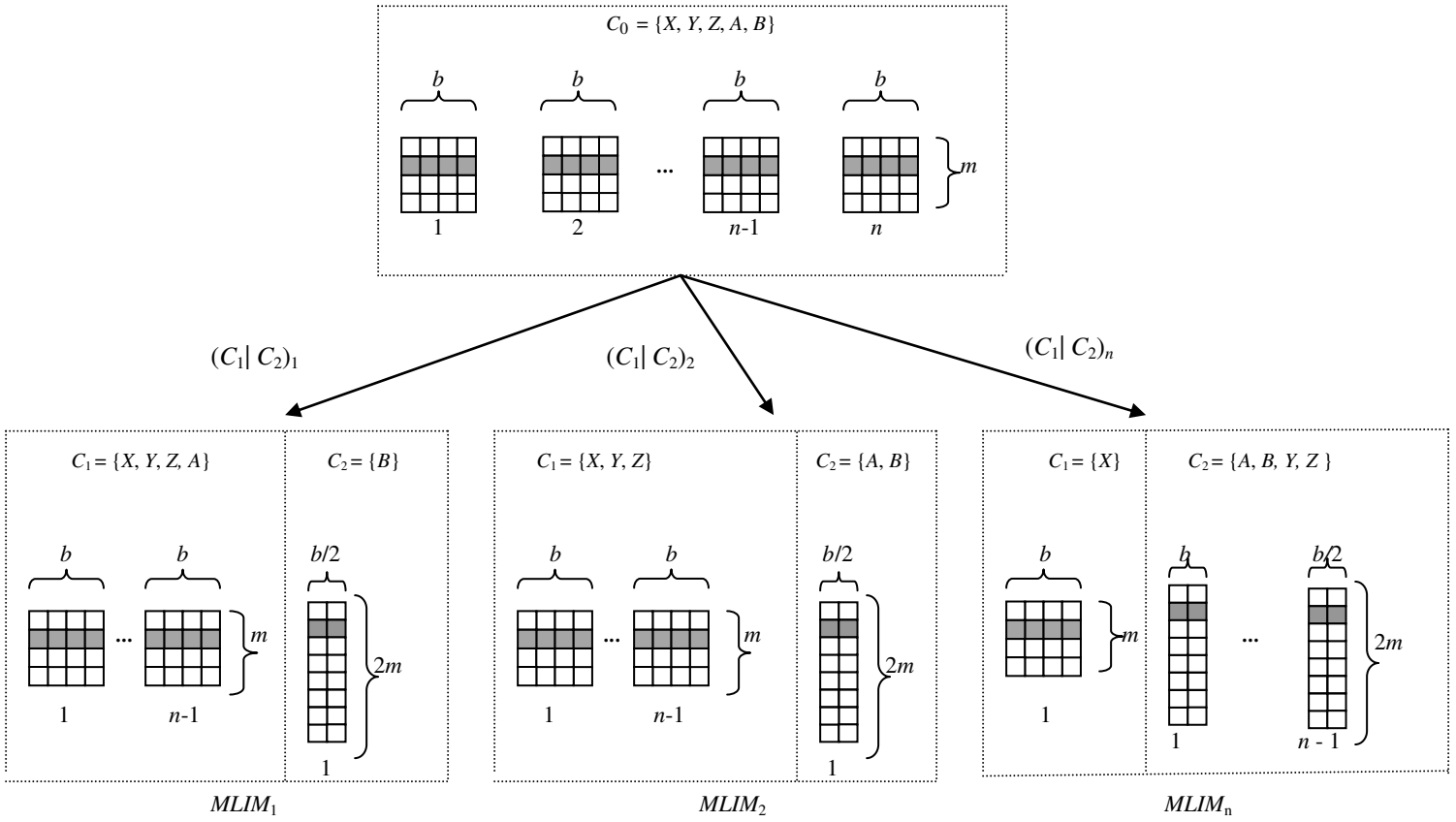


Figura 4.8 Exemplo de pares de *caches* particionados ($C_1|C_2$).

Considerando uma aplicação com uma quantidade *na* de matrizes acessadas, para cada configuração de *cache* unificado é possível obter até (*na* - 1) configurações de *caches* particionados. Utilizando *MLIM*, são necessárias até 2×(*na* - 1) + 1 simulações para obtermos os *caches* particionados de menor tempo médio de acesso à memória. Numa busca exaustiva pelo espaço de projeto sem utilizar a métrica *MLIM*, seriam necessárias 2^{na} simulações.

4.2.5 Fase de *Simulação de Cache II*

Uma vez que o *cache* particionado ($C_1|C_2$) e o mapeamento de matrizes em C_1 e C_2 foram determinados, o mesmo rastro usado para simular C_0 na fase de *Simulação de Cache I* é usado para simular C_1 e C_2.

Os *caches* particionados ($C_1|C_2$) que possuem mais de um mapeamento de matrizes precisam ser simulados para cada mapeamento porque podem ter diferentes quantidades de acertos e faltas no *cache*. Logo, a melhor configuração de *cache* particionado é especificada por ($C_1|C_2$) e pelo mapeamento de matrizes em C_1 e C_2.

Os resultados da simulação deste passo do algoritmo são:

- o número de *words* referenciadas em C_1 e C_2, indicado por $NREF_1$ e $NREF_2$, respectivamente;
- o número de blocos distintos referenciados pelas matrizes mapeadas por C_1 e C_2, designados por $N_t(C_1)$ e $N_t(C_2)$, respectivamente;
- o número de faltas nos *caches* C_1 e C_2, denominados NF_1 e NF_2, respectivamente.

4.2.6 Fase de *Avaliação do Cache*

Após o particionamento de *caches*, selecionamos o *cache* de matrizes particionado com o menor tempo médio de acesso à memória e comparamos com o *cache* de matrizes unificado de menor tempo médio de acesso à memória.

Em seguida, comparamos estes dois *caches* em termos de tempo de acesso, consumo de energia e taxa de faltas. Estas métricas são computadas dos resultados das simulações, conforme as Equações 4.6, 4.7 e 4.8 para um *cache* associativo C_i com i = 0, 1 e 2 [7] , [17].

$$Taxa\_de\_falta(C_i) = NF(C_i) / NREF_i \qquad \text{(eq. 4.6)}$$

$$Tempo\_médio\_acesso\_à\_memória(C_i) = Tempo\_de\_acerto(C_i) + Taxa\_de\_falta(C_i) \times Penalidade\_da\_falta(C_i) \qquad \text{(eq. 4.7)}$$

$$Energia\_média\_consumida(C_i) = Energia\_de\_acerto(C_i) + Taxa\_de\_falta(C_i) \times Penalidade\_de\_energia(C_i) \quad \text{(eq. 4.8)}$$

Para uma determinada tecnologia de circuito integrado, o tempo e energia de acerto, ou seja, o tempo e energia necessários para acessar o *cache*, respectivamente, dependem do tamanho de *cache* e de sua organização.

Por outro lado, a penalidade da falta e a penalidade da energia, ou seja, o tempo e a energia necessários para trazer uma nova linha no *cache*, dependem da organização de *cache* e dos outros níveis da hierarquia de memória, tais como o *cache* nível 2 e a memória principal.

O tempo e a energia de acerto podem ser estimados com base na implementação de *caches* usando ferramentas CAD. As estimativas usadas neste trabalho foram obtidas com a ferramenta CACTI [32], que fornece tempos e energias de acerto baseados em modelos de *cache* com várias organizações para diversas tecnologias de implementação.

A penalidade de tempo e energia decorrente da falta é estimada, neste trabalho, a partir do tamanho da linha e dos dispositivos usados para compor a memória principal.

Para um *cache* particionado ($C_1|C_2$), o cálculo da taxa de falta, tempo médio de acesso à memória e energia média consumida são modificados para considerar a percentagem de acessos a cada partição e a possível concorrência de acessos, conforme mostrado nas Equações 4.10, 4.11 e 4.13.

As equações 4.10 a 4.13 são médias ponderadas das frações f_1 e f_2 do total de acessos que são mapeados à C_1 e C_2, respectivamente, como determinado pela Equação 4.9, onde $f_1 + f_2 = 1$.

A Equação 4.11 assume que no máximo dois acessos paralelos ocorrem e que um deles é mapeado para C_1 e o outro para C_2.

A Equação 4.12 também assume no máximo dois acessos paralelos, mas considera uma fração P_i de acessos paralelos que são mapeados para a mesma partição de *cache* C_i, onde $i = 1, 2$ e $P_1 + P_2 \leq 1$.

$$f_i = NREF_i / NREF_0, \qquad i = 1,2 \quad \text{(eq. 4.9)}$$

$$Taxa\_de\_falta(C_1 \mid C_2) = f_1 \times Taxa\_de\_falta(C_1) + f_2 \times Taxa\_de\_falta(C_2) \quad \text{(eq. 4.10)}$$

$$\begin{aligned} &Tempo\_médio\_de\_acesso\_à\_memória(C_1 \mid C_2) = \\ &\quad f_1 \times Tempo\_médio\_de\_acesso\_à\_memória(C_1) \\ &+ f_2 \times Tempo\_médio\_de\_acesso\_à\_memória(C_2) \end{aligned} \qquad \text{(eq. 4.11)}$$

$$\begin{aligned} &Tempo\_médio\_de\_acesso\_à\_memória(C_1 \mid C_2) = \\ &\quad f_1 \times [Tempo\_médio\_de\_acesso\_à\_memória(C_1) + P_1 \times Tempo\_de\_acerto(C_1)] \\ &+ f_2 \times [Tempo\_médio\_de\_acesso\_à\_memória(C_2) + P_2 \times Tempo\_de\_acerto(C_2)] \end{aligned} \qquad \text{(eq. 4.12)}$$

$$\begin{aligned} &Energia\_média\_consumida(C_1 \mid C_2) = \\ &\quad f_1 \times Energia\_média\_consumida(C_1) \\ &+ f_2 \times Energia\_média\_consumida(C_2) \end{aligned} \qquad \text{(eq. 4.13)}$$

A vantagem de usar *caches* particionados quando existem acessos paralelos pode ser quantificada assumindo que no máximo dois acessos paralelos ocorrem, e uma fração Φ de acessos são paralelos com um acesso mapeado para C_1 e o outro para C_2.

Assumindo um *cache* de matrizes unificado com somente uma porta de *cache* para atender uma fração Φ de acessos simultâneos, o termo $(\Phi \times tempo\_acerto(C))$ deve ser adicionado à Equação 4.6, conforme mostrado na Equação 4.14 a seguir.

Este acréscimo ocorre porque o *cache* de matrizes unificado só consegue satisfazer um acesso por vez. Conseqüentemente, um dos acessos simultâneos é obrigado a esperar que o *cache* de matrizes unificado atenda ao outro acesso que é requisitado ao mesmo tempo que ele.

$$\begin{aligned} &Tempo\_médio\_de\_acesso\_à\_memória(C) = \\ &\quad (1+\Phi) \times Tempo\_de\_acerto(C) + Taxa\_de\_falta(C) \times Penalidade\_da\_falta(C) \end{aligned} \qquad \text{(eq. 4.14)}$$

As fases do nosso algoritmo, com suas respectivas entradas e saídas, estão resumidas na Figura 4.9.

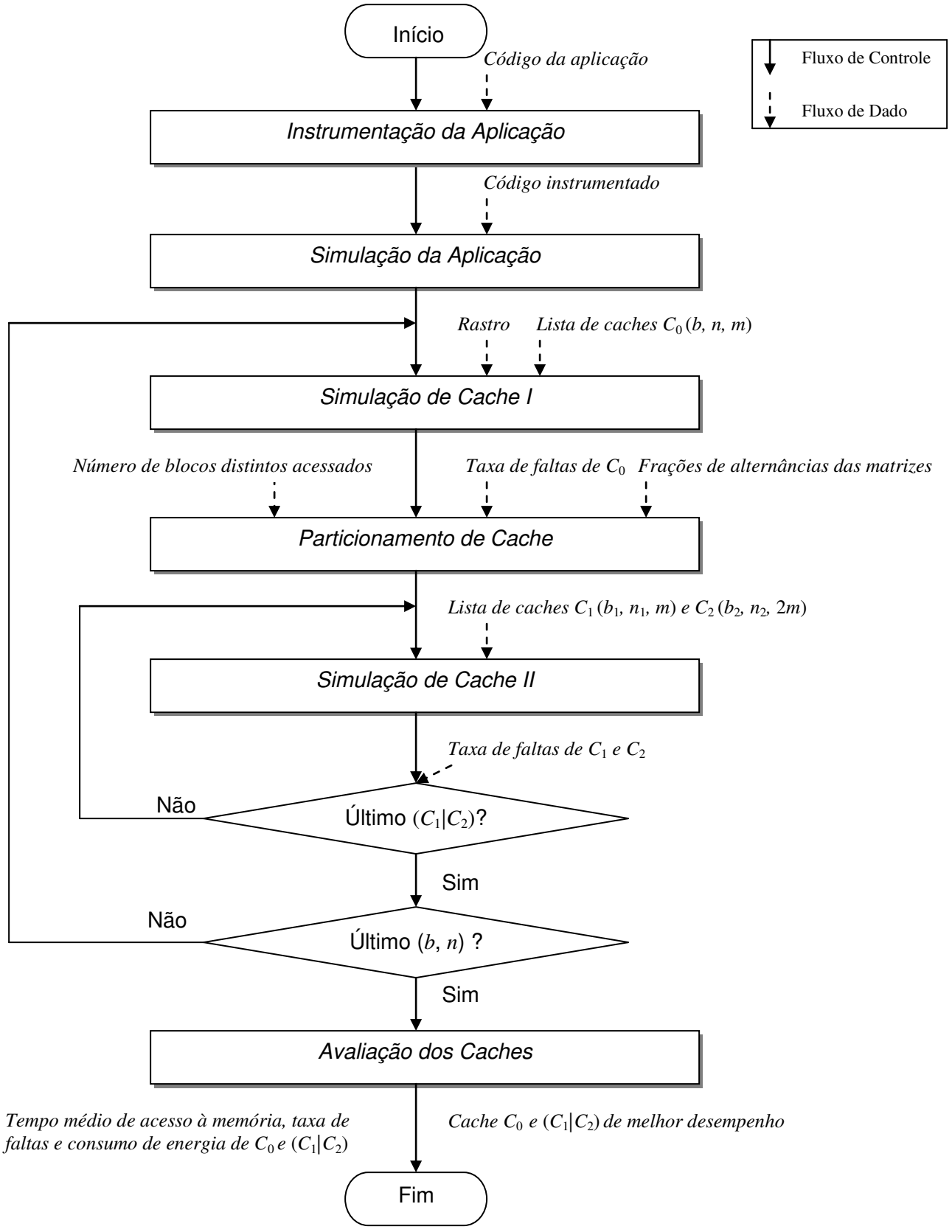


Figura 4.9 Etapas do algoritmo de particionamento de *caches* de matrizes nível 1.

4.2.7 Particionamento Iterativo de *Cache*

Suponha que uma matriz *A* apresente a seqüência de acessos sobre o bloco *B* de tamanho *b* mostrada na Figura 4.10, onde as *words* acessadas estão sombreadas.

Na Figura 4.10, *M*(*B*) é igual a 0, pois, em todos os acessos, de 1 a 4, somente a segunda metade do bloco é acessada. Pelo valor de *M*(*B*) sabemos que apenas metade da extensão do bloco *B* é efetivamente utilizada, mas não conseguimos saber que somente 1/8 do bloco está sendo acessado.

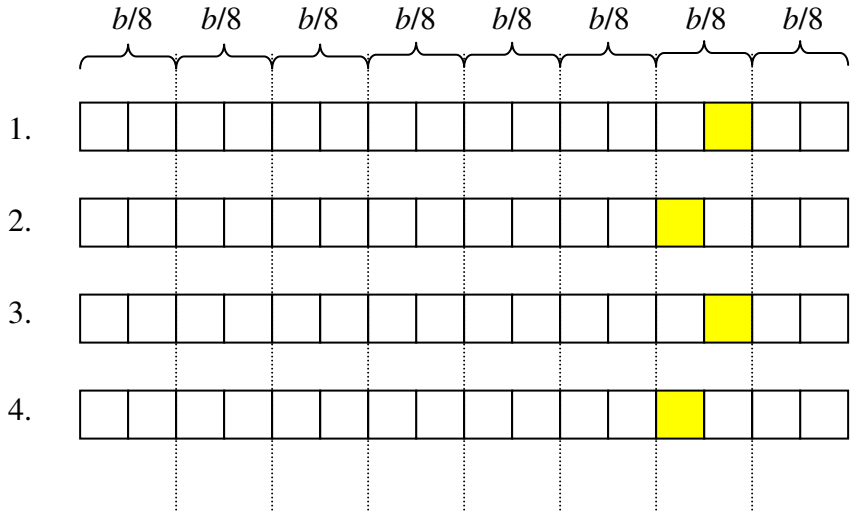


Figura 4.10 Acessos restritos à 1/8 da extensão do bloco *B*.

Para permitir o ajuste do tamanho de linha às aplicações que podem usar tamanhos de linha menores que a metade do comprimento de *B*, podemos aplicar nosso algoritmo iterativamente, conforme descrito no pseudocódigo da Figura 4.11.

O particionamento iterativo de *caches* considera como entrada qualquer uma das partições *i* do *cache* particionado produzido na iteração anterior, contanto que nenhuma das condições a seguir, descritas na linha 13, seja verdadeira:

- n_i seja igual a 1, quando não é mais possível dividir o *cache*;
- b_i seja igual a 1 *word*, quando o menor tamanho de linha de *cache* de matrizes unificado é alcançado;
- a quantidade de matrizes *na* mapeadas para C_i é igual a 1, quando se esgotam as possibilidades de atribuir matrizes a mais de um *cache*.

Cada nova iteração gera na linha 14 um novo par ($C_{i1}|C_{i2}$). Se o seu tempo médio de acesso à memória for menor que o do particionamento anterior, na linha 16 este novo par substitui a partição *i* do *cache particionado*, que corresponde ao *cache* particionado atual com todas as partições. Além disso, como um novo par ($C_{i1}|C_{i2}$) foi incluído no *cache particionado*, o número máximo de partições i_{max} é incrementado na linha 18.

O algoritmo prossegue particionando o *cache* iterativamente, até que o número de partições *i* a serem re-particionadas seja maior que i_{max}. Assim, ambas as partições podem ser usadas em cada aplicação iterativa do algoritmo, de modo que tanto C_{i1} quanto C_{i2} podem ser novamente particionados.

```
1   Particione_Cache_Iterativamente(b, n, m, M(A) para todas matrizes A, na)
2   // onde cache particionado é cache particionado atual com todas as partições
3   // i é a partição a ser re-particionada, b, n, m são os parametros desta partição

4   i = 0;
5   b_i = b;
6   n_i = n;
7   m_i = m;
8   M_i(A) = M(A);
9   na_i = na;
10  i_max = 0;
11  cache particionado = C_0;

12  enquanto (i ≤ i_max)
13      se ( (n_i > 1) , (b_i > 1 word) e (na_i > 1) )
14          Particione_Cache(b_i, n_i, m_i, M_i(A) para todas as matrizes A);
15          se ( tempo médio de acesso à memória < que o do particionamento anterior )
16              substitua a partição i do cache particionado por (C_i1|C_i2);
17              atualize os valores de b, n, m, M(A) e na para as duas novas partições i e i+1;
18              i_max++;
19          senão
20              i++;
21      senão
22          i++;
23  Fim de Particione_Cache_Iterativamente
```

Figura 4.11 Procedimento iterativo de particionamento de *caches*.

Desta forma, no exemplo da Figura 4.9, na primeira iteração do algoritmo, C_2 teria uma linha de tamanho *b*/2. Na segunda iteração do algoritmo, o *cache* C_{22} gerado a partir de C_2 teria uma linha de tamanho *b*/4. Por fim, na terceira iteração do algoritmo, o *cache* C_{222} gerado a partir de C_{22} teria uma linha de tamanho *b*/8.

Na Figura 4.12 ilustramos um exemplo de particionamento iterativo de *caches*. A partir de um *cache* de matrizes unificado com 6 vias de m conjuntos de linha de tamanho b, são gerados um *cache* C_1, com 2 vias de m conjuntos de linha de tamanho b, e um *cache* C_2, com 4 vias de $2m$ conjuntos de linha de tamanho $b/2$. Na segunda iteração do algoritmo, o *cache* C_2 gerado na iteração anterior é novamente particionado em um *cache* C_{21} com 3 vias de $2m$ conjuntos de linha de tamanho $b/2$ e um *cache* C_{22} com 1 via de $4m$ conjuntos de linhas de tamanho $b/4$.

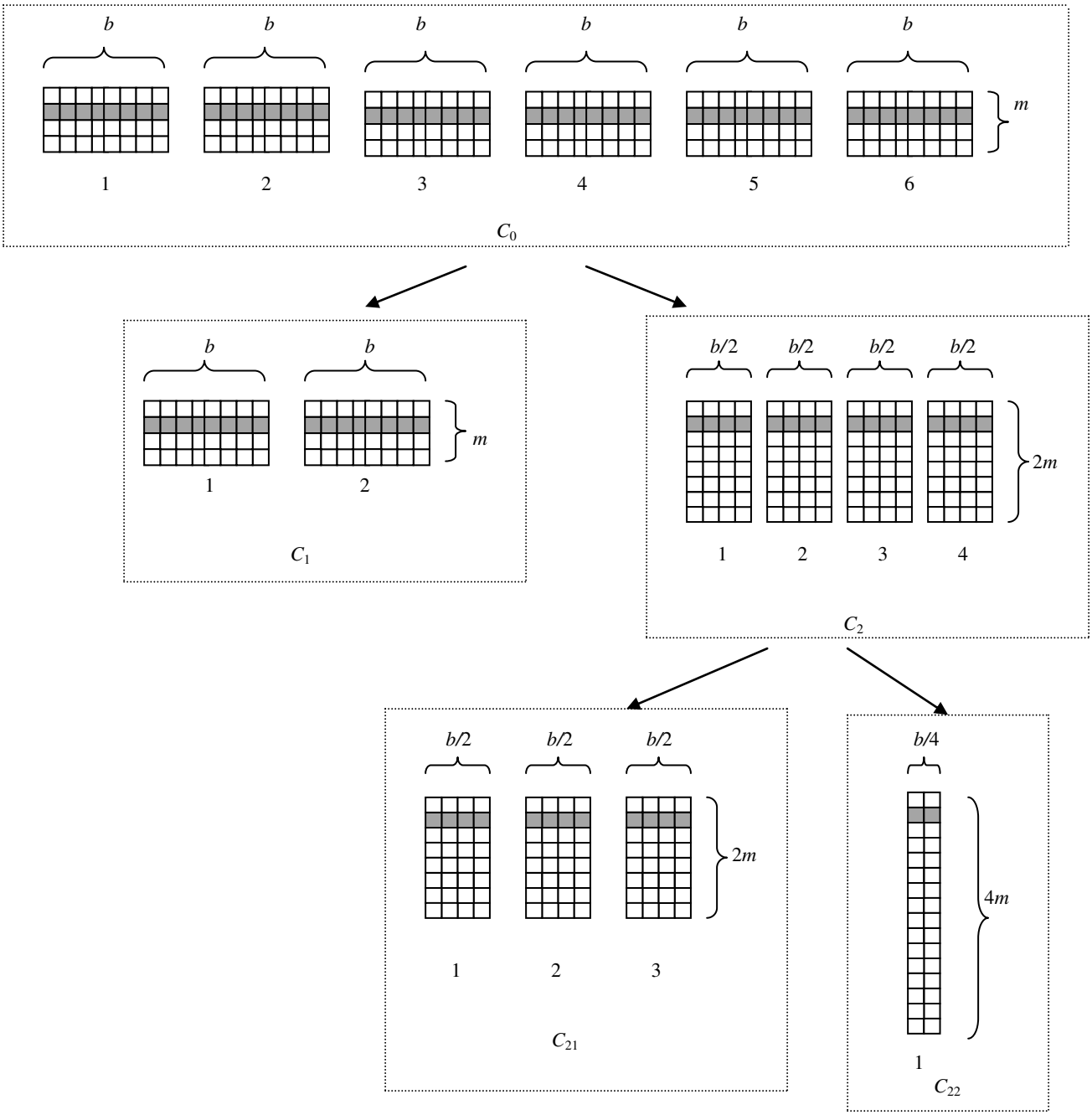


Figura 4.12 Exemplo de particionamento iterativo de *caches*.

O melhor desempenho obtido pelos *caches* de matrizes particionados em relação aos unificados é geralmente decorrente do acesso simultâneo às partições de *cache* e da redução no tamanho de *cache*, que leva a reduções no tempo de acerto e penalidade da falta. Quando maior for o número de partições de *cache*, menor o tamanho de *cache*, de linha e associatividade, e maior a quantidade de partições oferecendo acessos simultâneos em execuções paralelas.

No entanto, a taxa de faltas pode aumentar com o particionamento iterativo. Se o aumento for grande, o tempo médio de acesso à memória pode piorar, pois, conforme definido na Equação 4.7, a taxa de falta é multiplicada pela penalidade da falta.

O particionamento iterativo de *caches* pode ser particularmente interessante:

- em *caches* de matrizes unificados C_0 com valores de b e n altos, por permitir o ajuste da aplicação às diversas possibilidades de b_i e n_i;
- em aplicações com muitos valores distintos de $M(A)$, para explorar as possibilidades de atribuições de matrizes A a cada partição de *cache*;
- quando alguma das taxas P_1 ou P_2 de acessos paralelos a uma das partições na Equação 4.12 é alta.

4.3 Arquitetura de *Caches* Particionados

Nesta seção descrevemos as modificações de *hardware* necessárias para a implementação dos *caches* de matrizes particionados.

Existem diversas maneiras de implementar os *caches* de matrizes particionados. Para cada acesso à memória, o controlador de *cache* precisa gerar sinais adicionais que ativem C_1 ou C_2. Por exemplo, um decodificador pode receber como entrada a partição de *cache* mapeada e ativar o acesso à partição de *cache* acessada conforme mostrado na Figura 4.13, onde a saída do decodificador ativa a partição adequada do *cache* no controlador de *cache.* Logo, existe um aumento no tempo de acerto do *cache* devido à necessidade de gerenciar *caches* com mais de uma partição.

Para cada implementação, a entrada deste decodificador é gerada de maneira diferente.

É possível obter este sinal decodificando alguns dos *bits* do endereço do dado. O número de *bits* necessários para distinguir os acessos a C_1 dos a C_2 é normalmente bem pequeno devido aos seguintes motivos. Primeiro, os endereços dos dados assumem valores no espaço de endereços destinado a dados, sendo, portanto, valores delimitados. Segundo, estruturas de dados individuais podem possuir

um grande número de *words*. Por último, matrizes com endereços seqüenciais podem estar mapeadas na mesma partição de *cache*. Logo, o nível adicional de decodificação pode ser bem rápido.

Outra possibilidade é incluir a partição de *cache* em instruções de *load* e *store* customizadas. Esta modificação consiste em incluir nas instruções a indicação da partição de *cache* que deve ser consultada, baseado no mapeamento de matrizes às partições de *cache* produzido por nosso algoritmo. Esta informação é compilada junto com o código fonte e inserida no código *final*. Nas subseções 4.3.1 e 4.3.2 propomos 2 formas diferentes para realizar esta modificação. Estas formas estão relacionadas ao modo como a informação da partição de *cache* a ser acessada é armazenada nas instruções customizadas e usada durante a execução. Em ambas as propostas, o modo de funcionamento da *pipeline* permanece inalterado.

Por fim, é possível utilizar os *bits* do contador de programa (*PC*) para distinguir acessos de matrizes mapeados para C_1 ou C_2. Uma forma de realizar esta modificação é descrita na subseção 4.3.3.

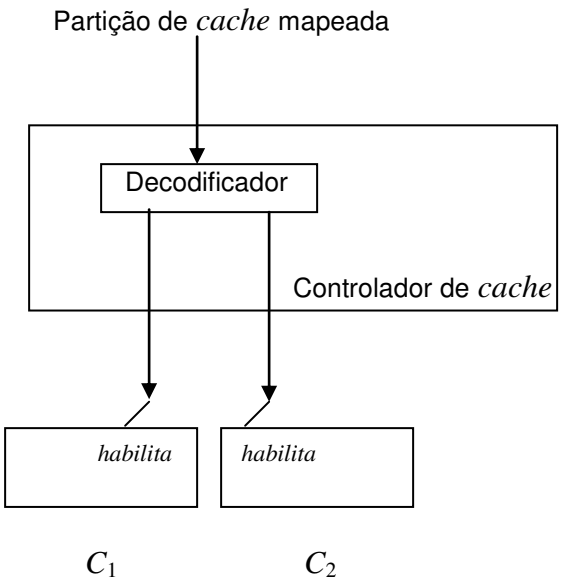


Figura 4.13 Modificações no controlador de *caches*.

4.3.1 Instruções com indicação da Partição de *Cache*

O *cache* é consultado toda vez que há acesso a um endereço de memória. Este endereço de memória pode ser acessado para leitura ou escrita. Quando ele é lido, a instrução executada é a de *load*. Quando ele é escrito, a instrução executada é a de *store*.

Para um processador com conjunto de instruções específico para uma aplicação, é possível usar instruções de *load* e *store* especiais, com *bits* reservados no código destas instruções para especificar a partição de *cache* correspondente ao dado associado. Ou seja, nesta primeira proposta, a informação armazenada nas instruções especiais é a partição de *cache* a ser acessada. A Figura 4.14 exemplifica estas instruções de *load* e *store.*

```
load registrador, endereço, partição de cache
store registrador, endereço, partição de cache
```

Figura 4.14 Exemplo de instrução *load* e *store* com mapeamento de partição de *cache*, onde a partição de *cache* a ser consultada em cada endereço de memória é armazenada no registrador das instruções de *load* e *store*.

A desvantagem desta proposta está na falta de flexibilidade para modificar o mapeamento da partição de *cache* em tempo de execução, visto que a indicação da partição do *cache* está no código da instrução.

4.3.2 Instruções com identificador da matriz e Tabela de Partição de *Cache*

Da mesma forma que a abordagem anterior, esta abordagem utiliza instruções de *load* e *store* especiais, com a diferença que aqui a informação armazenada nas instruções especiais é um identificador da matriz acessada.

Para permitir alterações dinâmicas no mapeamento de matrizes às partições de *cache*, uma alternativa à implementação anterior é preencher as instruções especiais de *load* e *store* com um identificador da matriz acessada e construir uma tabela para contem a partição de *cache* para cada matriz, conforme mostrado na Figura 4.15.

A partir do identificador da matriz contido nas instruções especiais de *load* e *store*, procuramos na tabela de mapeamento a sua partição de *cache* correspondente. Deste modo, para alterar este mapeamento após a compilação do código fonte, basta modificar o valor da partição de *cache* na tabela de mapeamento de matriz e partição de *cache*.

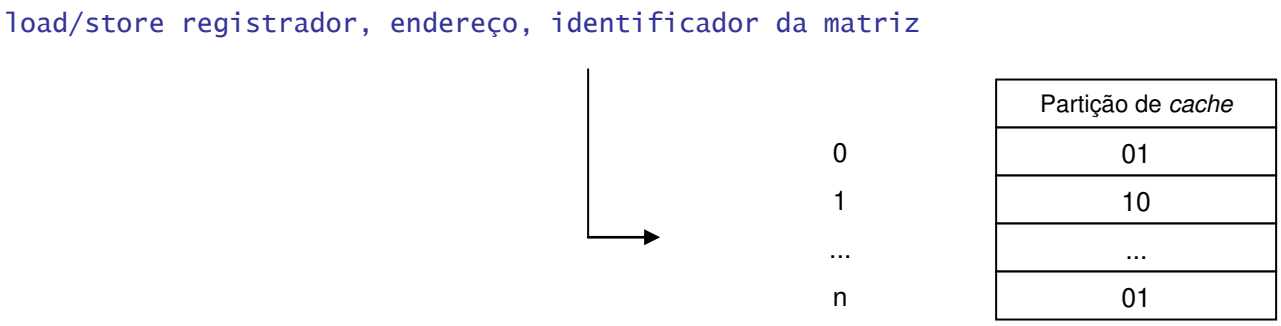


Figura 4.15 Exemplo de mapeamento do identificador da matriz à partição de *cache*.

4.3.3 Mapeamento do Endereço das Instruções às Partições de *Cache*

Podemos mapear o endereço de cada instrução de *load* e *store* às partições de *cache*.

Usamos bits do endereço da instrução para fazer acesso à tabela de mapeamento de partição, como exemplificado na Figura 4.16, usando apenas um número suficiente de bits para distinguir entre os endereços de instruções de acesso à memória. A vantagem desta abordagem é não precisar de instruções de *load* e *store* especiais, pois utiliza o endereço da instrução para encontrar a partição o *cache*.

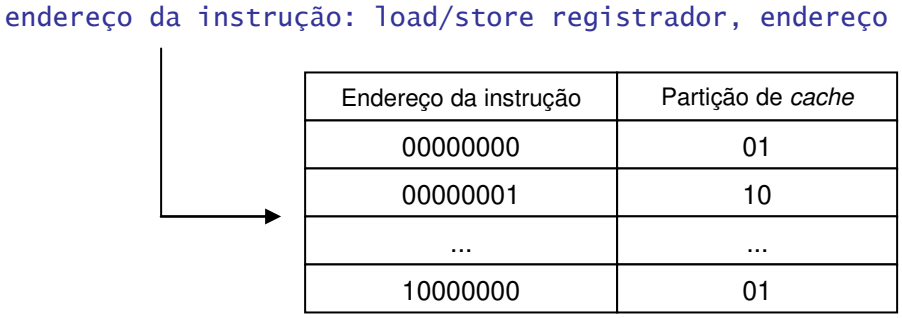


Figura 4.16 Exemplo de mapeamento de endereço da instrução à partição de *cache*.

Por outro lado, nesta abordagem é necessário procurar o endereço da instrução na tabela da Figura 4.16, o que pode aumentar ainda mais o tempo para acesso ao *cache*. A situação pode se tornar inviável ser for necessário usar um número grande de *bits* do endereço.

4.4 Conclusão

Neste capítulo apresentamos nosso algoritmo para geração de *caches* particionados e o correspondente mapeamento de matrizes com o intuito de reduzir o tempo médio de acesso à memória.

Apresentamos alternativas para implementar nosso algoritmo na arquitetura de *caches* particionados. O projetista do sistema embarcado pode escolher a alternativa mais adequada dependendo da quantidade de matrizes acessadas, do número de partições de *caches*, da quantidade de *bits* disponíveis para as alterações de *hardware* e da flexibilidade necessária para escolher a alteração de mapeamento de matrizes nas partições de *cache.*

A utilização de instruções especiais com mapeamento da partição de *cache* não aumenta muito o tempo de acerto do *cache* porque a partição de *cache* mapeada já está indicada no própria instrução, de modo que não é necessário procurar esta informação.

A alternativa de utilizar instruções e tabela com mapeamento da partição de *cache* aumenta pouco o tempo de acerto do *cache*, visto que implica em somente uma busca do identificador da matriz à partição de *cache*.

O mapeamento do endereço das instruções às partições de *cache*, por sua vez, pode elevar consideravelmente o tempo de acerto do *cache*.

O capítulo seguinte resume os resultados da aplicação do nosso algoritmo no particionamento de *caches* de matrizes.

Capítulo 5

Avaliação do Algoritmo

5.1 Introdução

Neste capítulo descrevemos o ambiente de execução do algoritmo e apresentamos as aplicações e os dados de entrada utilizados na avaliação do algoritmo. Descrevemos os *caches* de matrizes particionados obtidos, incluindo *caches* gerados em múltiplas iterações do algoritmo. Por fim, analisamos e comparamos nossos resultados com os de trabalhos anteriores.

5.2 Ambiente de Desenvolvimento e Simulação

Nosso algoritmo, métodos para instrumentar os acessos às matrizes das aplicações e simulador de *cache* foram implementados na linguagem de programação *C++* e desenvolvidos no Microsoft Visual Studio *C++* 2005. O ambiente de desenvolvimento consistiu de um microcomputador com sistema operacional Windows XP.

Devido à grande quantidade de simulações do nosso espaço de projeto, executamos nossas simulações paralelamente em 11 microcomputadores do Laboratório de Engenharia da Computação e Automação Industrial da Faculdade de Engenharia Elétrica da Unicamp. Estes microcomputadores possuíam o sistema operacional Windows.

5.3 Aplicações para Avaliação do Algoritmo

As aplicações usadas para a avaliação do algoritmo foram selecionadas segundo o critério abaixo:

- no mínimo 2 matrizes precisavam ser acessadas, para viabilizar a alocação de pelo menos uma matriz em cada um dos *caches* de matrizes particionados;
- o exame do código fonte não deveria revelar um padrão de acesso de matrizes que tornasse trivial a previsão da seleção de matrizes em cada partição de *cache.*

Cinco aplicações foram empregadas para demonstrar o uso do algoritmo desenvolvido neste trabalho: Convolução [28]; Transformada de Fourier (FFT) [29]; G3fax [42]; Codificador JPEG incluindo somente Transformada Inversa Discreta de Cosseno (IDCT) e quantização [30], [41]; e Decodificador de Vídeo MPEG-2 [31].

Para todas as aplicações, as entradas utilizadas foram escolhidas para permitir a geração de um rastro de acesso à memória que acessasse uma quantidade de blocos maior que a disponível no maior *cache* de matrizes unificado simulado neste trabalho.

A aplicação Convolução [28] calcula a integral do produto de duas funções f e g após uma delas ter sido refletida e deslocada. A aplicação Convolução é freqüentemente usada em processamento digital de sinais para detecção de pontos com descontinuidades no brilho [36]. Como os elementos das matrizes A da aplicação de Convolução são usados sequencialmente, seus valores de $M(A)$ são baixos. Entretanto, para matrizes com dimensões diferentes, os valores de $M(A)$ são distintos, pois uma matriz pode ocupar apenas meia linha. Para duas matrizes tais que ambas possuam dimensões muito grandes ou muito pequenas, os valores de $M(A)$ são iguais. Neste caso, bastaria um *cache* unificado. Por isso, em nosso trabalho, a aplicação Convolução linear foi simulada usando como entradas uma função com 8 e outra função com 3.500 pontos adjacentes. Para isso, alteramos o código fonte encontrado em [28] para atribuir o valor 8 à variável *lx*, e o valor 3.500 para a variável *ly*.

A aplicação Transformada de Fourier [29] é usada em processamento digital de sinais para transformar dados discretos do domínio de tempo para o domínio da freqüência. Em nosso trabalho, ela foi executada para implementar a transformada rápida de Fourier e sua transformada inversa sobre um vetor com 620 dados, pois a aplicação permite especificar o tamanho do vetor.

A aplicação G3fax [42] é um *benchmark* da Powerstone que decodifica um fax de formato digital do grupo 3, que segue as recomendações T.30 e T.4 da ITU-T [3]. Executamos a aplicação com uma

entrada de fax de 30.592 elementos. Como o tamanho das matrizes de entrada existentes no código fonte encontrado em [42] não era suficiente para acessar todos os blocos do maior *cache* de matrizes unificado simulado neste trabalho, aumentamos a quantidade de dados das matrizes de entrada realizando cópias dos dados originais.

Executamos a aplicação JPEG [30], [41] para um arquivo de 30K *bytes*. Após dividir a imagem em blocos de 8 x 8 *pixels*, a aplicação JPEG transforma cada bloco para o domínio coseno. Neste formato, como os valores do canto superior esquerdo dos valores transformados correspondem à essência da imagem, enquanto os valores do canto inferior direito correspondem aos detalhes, a precisão dos valores do canto inferior direito pode ser reduzida para comprimir a imagem sem comprometer sua qualidade. Em seguida, a imagem, codificada no domínio coseno, tem a sua precisão de *bit* reduzida para diminuir a quantidade de *bits* a serem codificados [41].

O decodificador de MPEG-2 [31] foi simulado para decodificar um arquivo de vídeo de 7K *bytes* em formato YUV. Este arquivo de vídeo, chamado *test.m2v*, pode ser encontrado em [31]. A decodificação foi configurada para usar precisão de 64 *bits* nos cálculos da transformada inversa de cosseno discreta e para armazenar os vídeos em *frames*.

Para cada uma das aplicações simuladas, a Tabela 5.1 detalha a linguagem de programação, o tamanho do código fonte, o número total de referências nos rastros, o número de matrizes acessadas, o tamanho das matrizes, o tamanho do espaço adicional para *caches* com linhas de 16, 32 e 64 *bytes*. O número total de referências nos rastros, número de matrizes, tamanho das matrizes e tamanhos do espaço adicional podem variar com as entradas usadas. Para cada aplicação, o tamanho de cada matriz acessada e o seu espaço adicional para *caches* com linhas de 16, 32 e 64 *bytes* está detalhado no Apêndice II.

Tabela 5.1 Características das aplicações usadas para avaliar o algoritmo.

Aplicação	Linguagem de Programação	Tamanho do Código Fonte (*bytes*)	Número Total de Referências nos Rastros	Número de Matrizes	Soma dos Tamanhos das Matrizes (*bytes*)	Tamanho do Espaço Adicional (*bytes*)		
						b = 64 *bytes*	*b* = 32 *bytes*	*b* = 16 *bytes*
Convolução	*C++*	5240	43692	3	28004	92	28	12
FFT	*C++*	28019	426356	4	180504	168	72	24
G3fax	*C*	30882	325518	4	59264	64	0	0
JPEG	*C*	3198	464972	10	16993	95	31	15
MPEG	*C*	192130	628807	49	91357	1123	419	131

5.4 Parâmetros Usados na Execução do Algoritmo

Nesta seção detalhamos os valores utilizados para os parâmetros do *cache* de matrizes unificado e para a avaliação dos *caches* de matrizes particionados. Explicamos também como foram gerados os rastros de acesso à memória após as aplicações terem sido instrumentadas.

5.4.1 Parâmetros do *Cache* de Matrizes Unificado

Nosso algoritmo foi avaliado em *caches* de 8 e 12K *bytes*. Estes tamanhos de *cache* foram escolhidos porque diversos tamanhos de *caches* embarcados se encontram neste intervalo, como os *caches* da famílias de processadores ARM9E, ARM10E e ARM11 [23]. Além disso, utilizamos *caches* de 12K *bytes* para avaliarmos o algoritmo de particionamento com *caches* com associatividades diferentes de potência de 2, como 3 e 6, por exemplo.

Consideramos linhas de 16, 32 e 64 *bytes*. Linhas de 32 e 64 *bytes* foram escolhidos devido ao menor tempo médio de acesso à memória observado nestes tamanhos de linhas em *caches* de 4 e 16K *bytes* [7]. Linhas de 16 *bytes* foram usados para fornecer mais uma opção de tamanho de linha.

Devido ao elevado custo de cada via, foram considerados *caches* embarcados com até 8 vias. Como um *cache* de matrizes unificado de n vias pode ser decomposto em até (n-1) *cache* particionados ($C_1|C_2$), e como usamos valores de associatividades iguais à 2, 3, 4, 6 e 8 nos *caches* de matrizes unificados, temos de 2-1=1 a 8-1=7 opções de particionamento.

Utilizamos estes valores pré-determinados de tamanhos de *cache*, tamanhos de linha e associatividade somente para limitar o espaço de projeto a ser explorado nas aplicações de avaliação, mas não há limite para estes valores impostos por nossa metodologia.

5.4.2 Parâmetros para Geração dos Rastros de Acesso à Memória

Assim que o rastro é gerado para uma entrada da aplicação, ele é usado como entrada no algoritmo para verificar se ele acessa uma quantidade de blocos distintos maior que a disponível pelo *cache* de 12K *bytes*. Esta primeira execução do rastro é sempre realizada para uma configuração de *cache* de 12K *bytes* porque, como o mesmo rastro é empregado nos *caches* de 8 e 12K *bytes*, se o rastro acessar um número de blocos distintos maior que o disponível no *cache* de 12K *bytes*, ele certamente acessará mais blocos que o existente no *cache* de 8K *bytes*.

Apenas uma simulação em alguma configuração do *cache* de 12K *bytes* é necessária, pois, para um mesmo valor de *b*, diferentes valores de *n* e *m* resultam no mesmo volume de linhas de *cache* disponíveis. Além disso, para diferentes valores de *b*, o número de blocos distintos acessados pelo rastro tende a variar na mesma proporção que o número de linhas disponíveis.

Garantimos que os acessos da aplicação inteira não cabem no maior tamanho de *cache* simulado, permitindo a ocorrência de faltas ao *cache* decorrentes não somente de faltas compulsórias e de capacidade. Quando esta condição não é satisfeita, o rasto gerado é descartado e a aplicação é executada novamente com novos dados de entrada selecionados para poder exercitar uma maior quantidade de blocos distintos, produzindo um rastro mais longo.

Nesta etapa, a fim de verificar se houve algum erro na instrumentação das aplicações, verificamos se o produto da quantidade de blocos distintos acessados pelo tamanho de linha é igual ou inferior ao tamanho da matriz declarado no código fonte.

5.4.3 Parâmetros para Avaliação de *Cache*

Os valores dos parâmetros das equações de avaliação de *caches*, descritos no Capítulo 4, foram pesquisados para os tamanhos de *cache*, linha e associatividades de todos *caches* de matrizes unificados e particionados simulados. Estes valores foram obtidos de diversas fontes.

Os valores do tempo e energia requeridos para um acerto foram extraídos do *software* CACTI disponível *online* [32], no modo normal, para uma tecnologia de 90 nm e um único banco de *cache*. Quando estes valores não estavam disponíveis no CACTI, estimamos o valor através de interpolação linear.

Para realizar esta interpolação linear, primeiro procuramos pela proporção encontrada em *caches* com mesmo tamanho total e tamanho de linha, mas com associatividades diferentes. Caso esta proporção não estivesse disponível, procuramos pela proporção encontrada em *caches* com tamanho imediatamente maior ou menor que o do *cache* avaliado, e mesmo tamanho de linha e associatividade. Quando esta proporção também não estava disponível, ou se ela já havia sido estimada por outros cálculos de proporção, procuramos pela proporção usada em outros tamanhos de linha, e mesmo tamanho de *cache* e associatividade.

Na Tabela 5.2(a), estimamos o valor do tempo de acerto para c=7, b=32 e n=3 com base nos valores do tempo de acerto para c=7, b=32, n=2 e c=7, b=32, n=4. Para os valores do tempo de acerto

para *c*=7, *b*=32 e *n*=5 a 8, usamos a proporção do aumento do tempo de acerto encontrados em *c*=8, *b*=32 e *n*=5 a 8 da Tabela 5.2(b), pois o CACTI não retornou um valor para a primeira configuração de *cache*, mas retornou um valor para a segunda.

Na Tabela 5.3(a), estimamos o valor do tempo de acerto para *c*=6, *b*=32 e *n*=5 a 8 com base nos valores do tempo de acerto para *c*=6, *b*=16, *n*=5 a 8, pois o CACTI não retornou um valor para *c*=6, *b*=32, *n*=8, mas retornou um valor para *c*=6, *b*=16, *n*=8.

Tabela 5.2 Exemplo de interpolação linear baseada na proporção encontrada em *caches* com tamanho *c* maior, e mesmo tamanho de linha *b* e associatividade *n*. Valores em negrito foram extraídos do CACTI [32]. Demais valores foram obtidos por interpolação.

(a)

b (*bytes*)	*n*	Tempo de acerto (ns)
c = 7K *bytes*		
32	1	**0,882080682**
32	2	**1,332600175**
32	3	1,385717757
32	4	**1,438835340**
32	5	1,541408912
32	6	1,643982483
32	7	1,746556055
32	8	1,849129627

(b)

b (*bytes*)	*n*	Tempo de acerto (ns)
c = 8K *bytes*		
32	1	**0,781606797**
32	2	**1,308343837**
32	3	1,346306577
32	4	**1,384269317**
32	5	1,482952915
32	6	1,581636513
32	7	1,680320111
32	8	**1,779003709**

Tabela 5.3 Exemplo de interpolação linear baseada na proporção encontrada em *caches* com tamanho de linha *b* menor, e mesmo tamanho de *cache c* e associatividade *n*. Valores em negrito foram extraídos do CACTI [32]. Demais valores foram obtidos por interpolação.

(a)

b (*bytes*)	*n*	Tempo de acerto (ns)
c = 6K *bytes*		
32	1	**0,794979217**
32	2	**1,315767966**
32	3	1,368834915
32	4	**1,421901864**
32	5	1,446540891
32	6	1,471179919
32	7	1,495818946
32	8	1,520457973

(b)

b (*bytes*)	*n*	Tempo de acerto (ns)
c = 6K *bytes*		
16	1	**0,804466958**
16	2	**1,377521356**
16	3	1,340956699
16	4	**1,304392042**
16	5	1,326994834
16	6	1,349597625
16	7	1,372200417
16	8	**1,394803208**

Os valores usados para a penalidade da falta levam em conta os acessos adicionais ao *cache* e à memória necessários para trazer um novo bloco ao *cache*. Os tempos de acesso da memória foram obtidos da especificação de uma memória DRAM266 [33] da Samsung™, segundo a qual são

necessários 67,5 ns para trazer a primeira *word* ao *cache* e 7,5 ns para trazer cada *word* adicional. Supomos que o barramento de memória tem largura suficiente para ler uma *word* de cada vez. Somamos a este valor de penalidade da falta o valor do tempo de acerto, pois o *cache* precisa ainda acessar o novo bloco inserido.

De acordo com [34], em microprocessadores e memórias comerciais típicas a penalidade de energia é de 50 a 200 vezes o valor da energia de acerto. Nosso trabalhou considerou que, para cada tamanho de *cache* e de linha, a penalidade de energia é 50 vezes a energia de acerto para um *cache* diretamente mapeado, para qualquer que seja a associatividade do *cache* avaliado. Consideramos o valor 50 para evitar a supervalorização da redução do tamanho de linha na redução da penalidade de energia, possibilitando a avaliação do nosso algoritmo sob o valor de penalidade de energia menos favorável à melhoria de consumo de energia.

Os valores de tempo de acerto, penalidade da falta, energia de acerto e penalidade da energia estão listados no Apêndice III.

Calculamos o tempo médio de acesso do *cache* de matrizes unificado considerando frações Φ de acessos paralelos iguais à 0; 0,25; 0,50; 0,75 e 1, sendo que os valores mais comuns de frações de acesso paralelos são 0 e 0,25.

5.5 Resultados do Particionamento de *Caches*

Nesta seção apresentamos os resultados do particionamento de *caches* apresentando os *caches* com menor tempo médio de acesso à memória. Estes *caches* são também avaliados segundo seu consumo de energia e taxa de falta.

Os *caches* foram obtidos através de 310 simulações do nosso algoritmo para 5 aplicações, diferentes configurações de *cache* de matrizes unificado e todos os valores de *MLIM*.

Os números de *caches* de matrizes unificados considerados são 9 *caches* de 8K *bytes* e 6 *caches* de 12K *bytes*. Conforme explicado no Capítulo 4, o número de *caches* de matrizes particionados simulados por *cache* de matrizes unificado necessários numa busca exaustiva é 2^{na}, onde *na* é o número de matrizes de uma aplicação embarcada. A Tabela 5.4 mostra o número de *caches* de matrizes particionados simulados para encontrar os *caches* de matrizes particionados com os menores tempos médios de acesso à memória usando *MLIM* e usando busca exaustiva. Notamos pela Tabela 5.4

que o uso de *MLIM* para buscar os *caches* de matrizes particionados com os menores tempos médios de acesso à memória diminuem consideravelmente o número de simulações necessárias.

Tabela 5.4 Número de *caches* de matrizes particionados simulados para encontrar *caches* de matrizes particionados com os menores tempos médios de acesso à memória usando busca exaustiva e *MLIM*.

Aplicação	Número de Matrizes	Número de *Caches* de Matrizes Particionados Usando Busca Exaustiva		Número de *Caches* de Matrizes Particionados Usando *MLIM*	
		c = 8K *bytes*	*c* = 12K *bytes*	*c* = 8K *bytes*	*c* = 12K *bytes*
Convolução	3	72	48	12	8
FFT	4	144	96	21	14
G3fax	4	144	96	12	8
JPEG	10	9216	6144	27	20
MPEG	49	5,066E+15	3,37E+15	107	81

Os números totais de referências acessadas, números totais de blocos distintos acessados e taxa de faltas de todos *caches* de matrizes unificados simulados da *Fase de Simulação I* do algoritmo são descritos no Apêndice IV. As frações de alternâncias de acessos nas metades da linha e os números de blocos distintos acessados por cada matriz em todos os *caches* de matrizes unificados simulados da *Fase de Simulação I* do algoritmo são descritos no Apêndice V. Os resultados da fase de *Particionamento de Caches*, *Simulação II* do algoritmo e *Avaliação de Caches* para todos os *caches* de matrizes particionados simulados são descritos no Apêndice VI.

5.5.1 Resultados: Tempo Médio de Acesso à Memória

Para cada aplicação, a Tabela 5.5 descreve as organizações de *cache* de matrizes unificado e particionado de menor tempo médio de acesso à memória para *caches* de 8 e 12K *bytes*. O *cache* particionado de menor tempo médio de acesso à memória não é necessariamente produzido a partir do *cache* de matrizes unificado de menor tempo médio de acesso a memória, pois ele é selecionado dentre todos os *caches* de matrizes particionados produzidos a partir de todos os *caches* de matrizes unificados simulados.

Para uma mesma aplicação, a quantidade de simulações do *cache* de 12K *bytes* é sempre menor que a do *cache* de 8K *bytes* porque no *cache* de 12K *bytes* só simulamos duas associatividades (3 e 6), ao passo que no *cache* de 8K *bytes* simulamos três (2, 4 e 8).

Observamos que, para todos os *caches* particionados da Tabela 5.5, existe pelo menos uma partição de *cache* diretamente mapeada. Isto é provavelmente devido ao baixo tempo de acerto de *caches* diretamente mapeados, que é de 50 a 70% do tempo de acerto dos *caches* associativos de 2 vias com mesmo tamanho de linha e de *cache* [32].

Notamos que, exceto pela aplicação MPEG, as frações de acesso f_1 e f_2 e a configuração de tamanho de linha dos *caches* de matrizes particionados tende a se manter aproximadamente igual nos *caches* de 8 e 12K *bytes*, o que indica que a mesma partição de matrizes foi usada.

Para as aplicações Convolução, FFT e JPEG, as configurações dos *caches* de matrizes particionados apresentam linhas de 32 e 64 *bytes.*

A Tabela 5.5 mostra que, quando o objetivo é reduzir o tempo médio de acesso à memória, a melhor forma de utilizar a área adicional de *cache* depende da aplicação e do número de partições de *cache*.

Tabela 5.5 *Caches* de matrizes unificados e particionados com os menores tempos médios de acesso à memória. Organização de *cache*: (tamanho de linha, associatividade, número de conjuntos).

Apl.	*Cache* de matrizes unificado C_0								*Cache* de matrizes particionado (C_1/C_2)				
	(b, n, m)	taxa de falta (x1000)	Tempo médio de acesso à memória (ns)					energia (nJ)	(b_1, n_1, m_1)| (b_2, n_2, m_2)	$f_1 \| f_2$	taxa de falta (x1000)	Tempo médio de acesso à memória (ns)	energia (nJ)
			Φ = 0	Φ = 0,25	Φ = 0,5	Φ = 0,75	Φ = 1						
Cache de 8K *bytes*													
Conv.	(64, 2, 64)	25,13	6,06	6,43	6,81	7,18	7,56	0,325	(64, 1, 64)| (32, 1, 128)	67,99| 32,01	25,11	5,28	0,173
FFT	(64, 2, 64)	10,07	3,32	3,70	4,07	4,45	4,82	0,255	(64, 1, 32)| (32, 3, 64)	9,52| 90,48	17,97	3,91	0,125
G3fax	(32, 4, 64)	3,30	1,79	2,14	2,48	2,82	3,17	0,098	(16, 1, 256)| (8, 1, 512)	78,78| 21,22	10,48	1,78	0,050
JPEG	(32, 2, 128)	1,59	1,50	1,83	2,16	2,48	2,81	0,063	(64, 1, 64)| (32, 1, 128)	90,73| 9,27	2,94	1,15	0,087
MPEG	(16, 8, 64)	2,33	1,61	1,96	2,31	2,66	3,01	0,080	(32, 1, 128)| (16, 1, 256)	47,79| 52,21	3,86	1,18	0,043
Cache de 12K *bytes*													
Conv.	(64, 3, 64)	24,90	6,19	6,61	7,03	7,45	7,86	0,407	(64, 2, 64)| (32, 1, 128)	67,99| 32,01	25,11	5,82	0,270
FFT	(16, 6, 128)	8,25	2,19	2,55	2,92	3,28	3,64	0,087	(64, 1, 64)| (32, 2, 128)	9,52| 90,48	5,78	2,14	0,081
G3fax	(32, 3, 128)	3,06	1,76	2,10	2,45	2,79	3,14	0,087	(32, 1, 64)| (16, 5, 128)	76,43| 23,57	4,31	1,33	0,069
JPEG	(32, 3, 128)	1,20	1,53	1,88	2,22	2,57	2,91	0,082	(64, 1, 64)| (32, 2, 128)	90,73| 9,27	1,85	1,07	0,087
MPEG	(32, 3, 128)	1,36	1,55	1,90	2,24	2,59	2,93	0,083	(64, 2, 64)| (32, 1, 128)	20,21| 78,79	1,88	1,22	0,078

Para as aplicações Convolução e JPEG, a melhor opção é aumentar somente a associatividade para ambos os *caches* de matrizes unificados e particionados. Para a aplicação FFT, a área adicional deve ser empregada como conjuntos de linhas adicionais, mesmo que em detrimento do tamanho de linha e associatividade. Para a aplicação G3fax, a área adicional deve ser explorada como conjuntos de linhas adicionais. Para a aplicação MPEG, a área adicional deve ser explorada como conjuntos de linhas adicionais, para *caches* de matrizes unificados, e para aumento do tamanho de linha, para os *caches* de matrizes particionados.

A Figura 5.1 compara os *cache* de matrizes unificados e particionados da Tabela 5.5, em termos de tempo médio de acesso à memória. Como esperado, a vantagem dos *caches* de matrizes particionados sobre os unificados aumenta com a fração $\boldsymbol{\Phi}$ de acessos paralelos, para todas as aplicações.

Nossas comparações entre *caches* de matrizes unificados e particionados dividem a diferença de valores encontrados entre *caches* de matrizes unificados e particionados pelo valor referente ao *cache* de matrizes unificado, conforme especificado pela Equação 5.1. Deste modo, mostramos o percentual de redução que o *cache* de matrizes particionados obtém sobre o *cache* de matrizes unificado.

$$Dif\,(Métrica) = \frac{\left[Métrica(C_0) - Métrica(C_1 \mid C_2)\right]}{Métrica(C_0)} \qquad \text{(eq. 5.1)}$$

Para a aplicação Convolução, a taxa de falta dos *caches* de matrizes particionados é próxima à do *cache* de matrizes unificado. Porém, como os *caches* de matrizes particionados reduzem o tamanho do *cache*, o tamanho de linha e a associatividade em relação ao *cache* de matrizes unificado, há redução no tempo médio de acesso à memória.

A aplicação FFT inclui vetores, nos quais os acessos são uniformemente distribuídos por toda a linha de *cache*, de modo que os *caches* de matrizes particionados não trazem reduções no tempo médio de acesso à memória, a não ser no caso de acessos paralelos.

As Tabelas 5.6 e 5.7 mostram a fração de alternância $M(A)$ nos acessos às metades das linhas das matrizes da aplicação G3fax nos *caches* de 8 e 12K *bytes*, respectivamente, enquanto a Tabela 5.8 mostra a quantidade $N(A)$ de blocos distintos acessados.

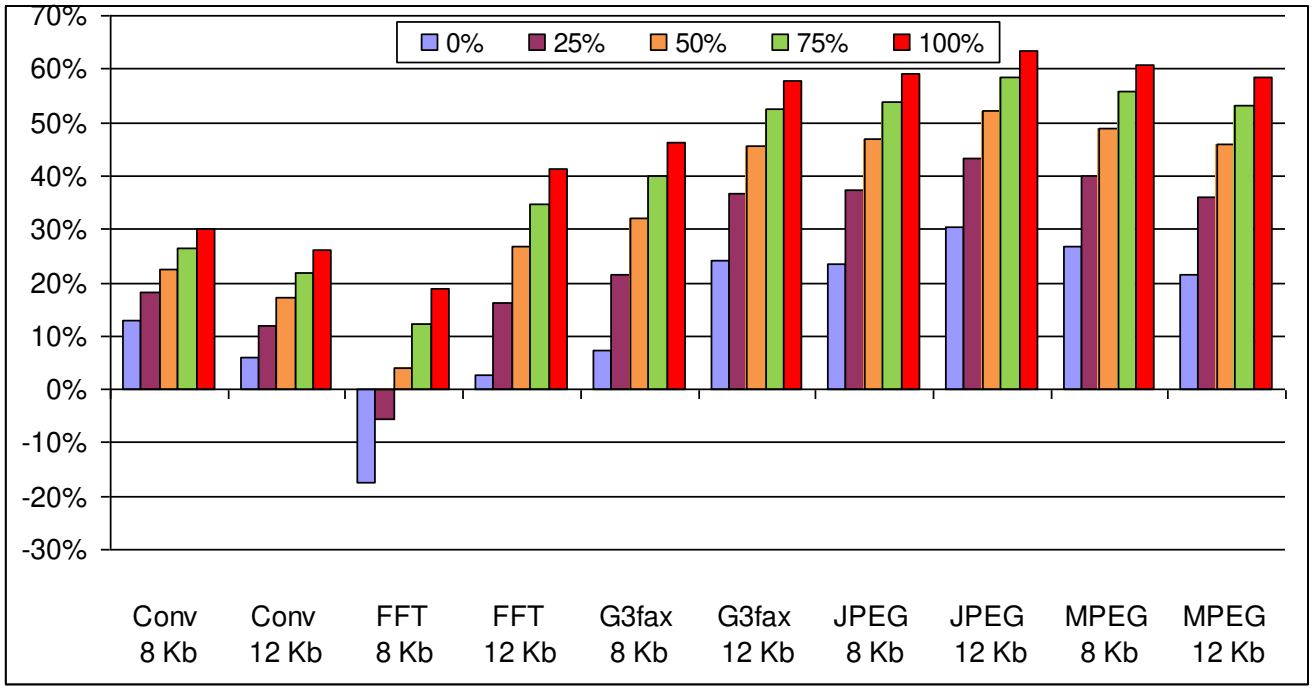


Figura 5.1 Redução no tempo médio de acesso à memória dos *caches* de matrizes particionados em relação aos *caches* de matrizes unificados para diferentes valores de Φ.

Observando as Tabelas 5.6 e 5.7, vemos que as matrizes *g3black* e *g3white* possuem $M(A)$ igual à zero para todas as configurações de *cache* simuladas, ou seja, a maioria dos blocos das matrizes *g3black* e *g3white* usam no máximo 8 *bytes* do tamanho da linha.

Olhando na Tabela 5.8 a quantidade de blocos distintos acessados por estas matrizes para b = 16 *bytes*, isto significa que no *cache* unificado a fração do espaço ocioso é no mínimo igual à [(N(*g3black*) + N(*g3white*)]/2) / (N(*fax*) + N(*g3black*) + N(*g3white*) + N(*rowbuf*)) = [(93+23)/2] / (955+93+23+108) = 58/1179 = 4,92%. Durante o particionamento de *caches*, este espaço inutilizado é transformado em linhas adicionais.

Tabela 5.6 $M(A)$ das matrizes da aplicação G3fax para *cache* de 8K *bytes*.

A	$M(A)$								
	b=64 *bytes*			b=32 *bytes*			b=16 *bytes*		
	n=8	n=4	n=2	n=8	n=4	n=2	n=8	n=4	n=2
fax	6,26	6,26	6,26	12,51	12,51	12,51	25,01	25,01	25,01
g3black	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
g3white	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
rowbuf	12,02	12,02	12,02	24,06	24,06	23,95	48,35	48,35	48,28

Tabela 5.7 *M*(*A*) das matrizes da aplicação G3fax para *cache* de 12K *bytes*.

A	*M*(*A*)					
	b=64 *bytes*		*b*=32 *bytes*		*b*=16 *bytes*	
	n=6	*n*=3	*n*=6	*n*=3	*n*=6	*n*=3
fax	6,26	6,26	12,51	12,51	25,01	25,01
g3black	0,00	0,00	0,00	0,00	0,00	0,00
g3white	0,00	0,00	0,00	0,00	0,00	0,00
rowbuf	12,02	12,02	24,06	24,06	48,39	48,39

Tabela 5.8 *N*(*A*) das matrizes da aplicação G3fax.

A	*N*(*A*)		
	b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
fax	240	478	955
g3black	86	91	93
g3white	15	18	23
rowbuf	28	55	108

As aplicações com as maiores melhorias são JPEG e MPEG porque o cálculo do IDCT causa uma concentração de acessos sobre os blocos de 8x8 de uma matriz de imagens.

5.5.2 Resultados: Taxa de Faltas e Consumo de Energia

A Figura 5.2 mostra a redução na taxa de falta dos *caches* de matrizes particionados de menor tempo médio de acesso à memória em relação aos *caches* de matrizes unificados de menor tempo médio de acesso à memória. Os valores negativos indicam que a taxa de faltas nos *caches* de matrizes particionados é, em muitos casos, maior que a do *cache* de matrizes unificado. A redução na taxa de falta para os *caches* de matrizes particionados de 12K *bytes* na aplicação FFT é resultado do mapeamento de matrizes com frações distantes de alternâncias de acessos às metades da linha de *cache*. Entretanto, exceto pela aplicação FFT, estas piores taxas de falta não impedem a redução no tempo médio de acesso à memória.

A Figura 5.3 ilustra a redução no consumo de energia dos *caches* de matrizes particionados de menor tempo médio de acesso à memória sobre os *caches* de matrizes unificados de menor tempo médio de acesso à memória. A menor associatividade das partições dos *caches* de matrizes particionados tende a reduzir a energia de acerto. Entretanto, alguns dos *caches* de matrizes

particionados selecionados podem ter linhas maiores que o dos *caches* de matrizes unificados de mesmo tamanho, e, conseqüentemente, podem requerer maiores energias de acerto e penalidade.

Para as aplicações Convolução e G3fax, para ambos os tamanhos de *caches*, e para a aplicação FFT no *cache* de 8K *bytes*, a redução no consumo de energia é devida à redução no tamanho de *cache* e no tamanho de linha dos *caches* de matrizes particionados, que proporcionam diminuições nas energias de acerto e de penalidade se comparadas com as dos *caches* de matrizes unificados. Embora o tamanho de linha dos *caches* de matrizes particionados da aplicação FFT no *cache* de 12K *bytes* seja maior que o do *cache* de matrizes unificado, há redução no consumo de energia por causa da redução da taxa de falta.

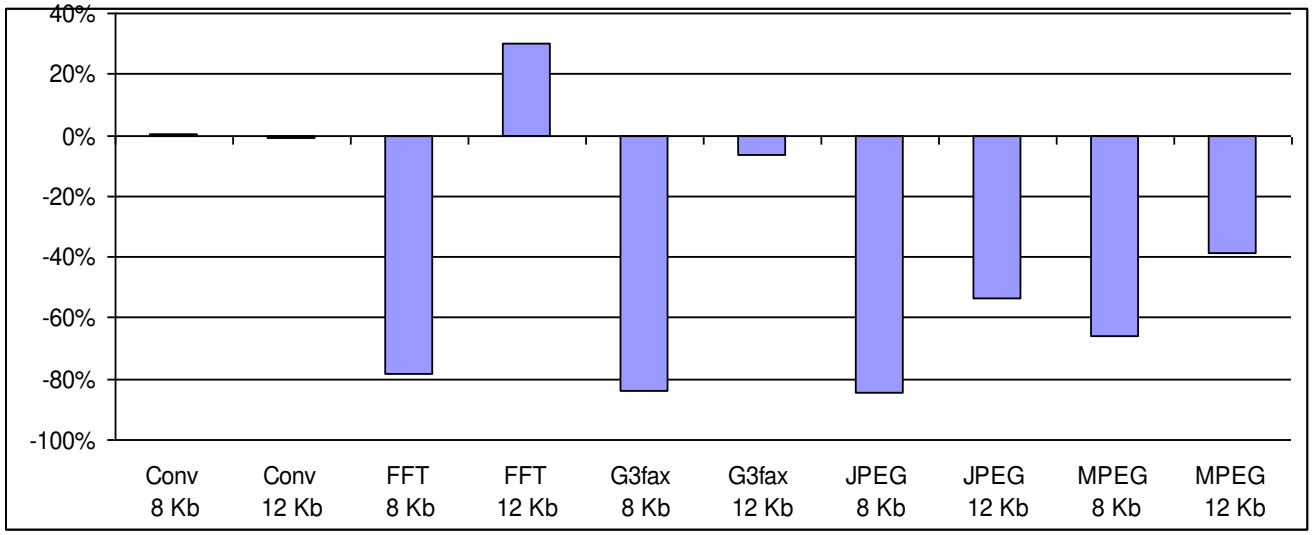


Figura 5.2 Redução na taxa de falta dos *caches* de matrizes particionados em relação aos *caches* de matrizes unificados. Valores negativos indicam que a taxa de faltas nos *caches* de matrizes particionados é, em muitos casos, maior que a do *cache* de matrizes unificado.

A redução no consumo de energia favorece os *caches* de matrizes particionados para todas as aplicações, exceto JPEG. Para a aplicação JPEG, a taxa de falta dos *caches* de matrizes particionados é maior que a do *cache* de matrizes unificado. Além disso, o tamanho de linha de uma das partições dos *caches* de matrizes particionados (64 *bytes*) é maior que a do *cache* de matrizes unificado (32 *bytes*).

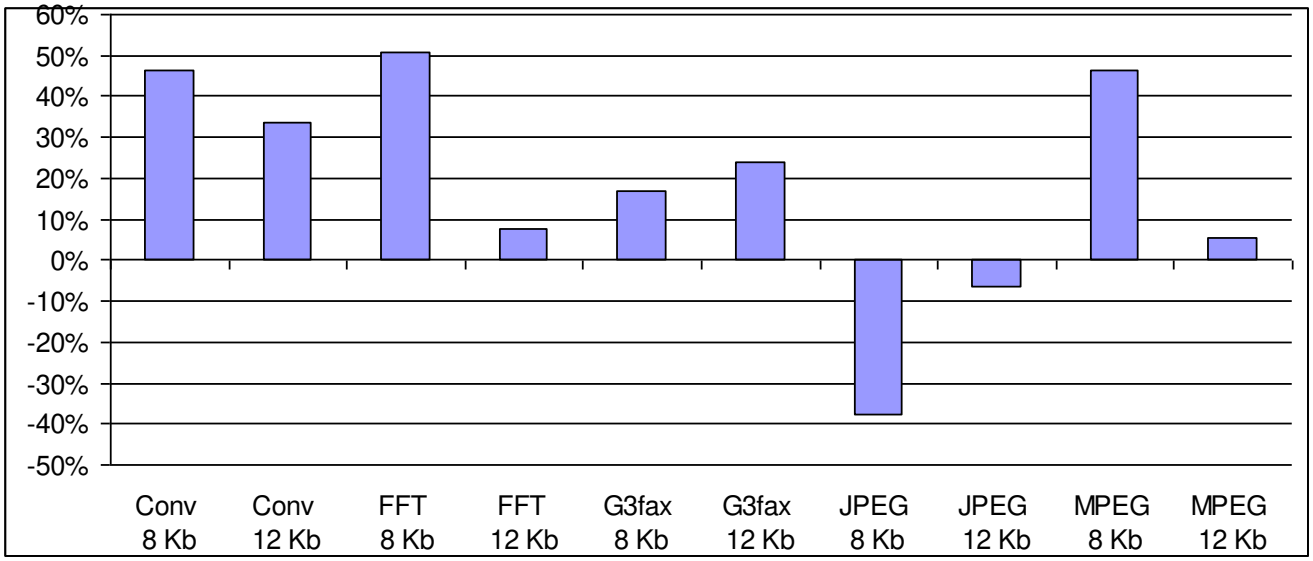


Figura 5.3 Redução no consumo de energia dos *caches* de matrizes particionados em relação aos *caches* de matrizes unificados. Valores negativos indicam que o consumo de energia pode aumentar quando os *caches* de matrizes particionados são selecionados segundo o menor tempo médio de acesso à memória.

Consultando a Tabela 5.9, vemos que o *cache* de matrizes unificado de 8K *bytes*, linhas de 32 *bytes* e 2 vias tem penalidade de 2,26 nJ. Seus *caches* de matrizes particionados de 4K *bytes* com linhas de 64 *bytes* e 1 via, e de 4K *bytes* com linha de 32 *bytes* e 1 via, possuem penalidades de 4,19 e 1,84 respectivamente. O *cache* de matrizes unificado de 12K *bytes*, linha de 32 *bytes* e 3 vias tem penalidade de 2,64 nJ. Seus *caches* de matrizes particionados de 4K *bytes* com linha de 64 *bytes* e 1 via, e de 8K *bytes* com linha de 32 *bytes* e 2 vias, possuem penalidades de 4,19 e 2,26 respectivamente. Neste caso, esta diferença na penalidade de energia, somada à piora na taxa de faltas causa um aumento no consumo de energia.

Tabela 5.9 Penalidade de energia em nJ por tamanho de *cache c* e tamanho de linha *b* [32].

b (*bytes*)	*c* (K *bytes*)		
	4	8	12
64	4,19	4,65	5,04
32	1,84	2,26	2,64

De qualquer modo, mesmo com *caches* de matrizes particionados selecionados somente segundo o tempo médio de acesso à memória, muitos dos *caches* de matrizes particionados também reduzem o consumo de energia se comparados aos *caches* de matrizes unificados de mesmo tamanho.

Os resultados do particionamento de *caches* considerando os *caches* de matrizes unificados e particionados selecionados pelo menor consumo de energia e menor produto do consumo de energia e tempo médio de acesso à memória são mostrados nos Apêndices VIII e IX, respectivamente.

5.6 Particionamento Iterativo de *Caches* e Resultados

Vemos na Tabela 5.5 que os *caches* C_2 com associatividade maior que um são encontrados nas aplicações FFT, G3fax e JPEG a partir do *cache* de matrizes unificado de 12K *bytes*, e na aplicação FFT a partir do *cache* de matrizes unificado de 8K *bytes*. O *cache* C_1 gerado para as aplicações de Convolução e MPEG a partir do *cache* de matrizes unificado de 12K *bytes* também possui associatividade maior que um.

Para estes *caches*, vemos pela Tabela 5.10 que o número de matrizes neles mapeados é maior ou igual a 2. Portanto, estes são os *caches* que podem ser usados numa segunda iteração do nosso algoritmo.

Tabela 5.10 Número de matrizes no *cache* de matrizes particionado utilizado na segunda iteração do algoritmo.

Aplicação	*c* (K *bytes*)	*Caches* de Matrizes Particionados Utilizado na Segunda Iteração	Número de matrizes no *Cache* de Matrizes Particionado Utilizado na Segunda Iteração
Convolução	12	C_1	2
FFT	8	C_2	2
FFT	12	C_2	2
G3fax	12	C_2	3
JPEG	12	C_2	4
MPEG	12	C_1	18

Entretanto, a Figura 5.4 e a Tabela 5.11 mostram que nenhum dos *caches* de matrizes particionados na segunda iteração obteve tempos médios de acesso à memória menores que os obtidos na primeira iteração. Isto não exclui a possibilidade de que melhores resultados possam ser obtidos se explorarmos todas as iterações de todos os *caches* de matrizes particionados, ao invés de aplicar iterativamente o algoritmo somente no *cache* de matrizes particionado com menor tempo médio de aceso à memória.

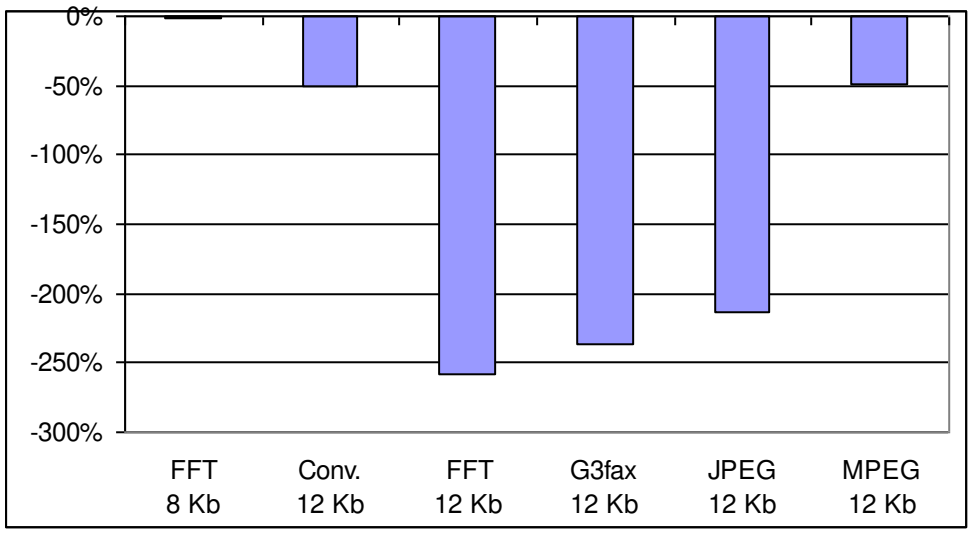


Figura 5.4 Percentagem da redução no tempo médio de acesso à memória na segunda iteração do algoritmo. Valores negativos indicam que o menor tempo médio de acesso à memória é alcançado já na primeira iteração do algoritmo.

Tabela 5.11 *Cache* de matrizes particionado da primeira e segunda iteração do algoritmo. Organização de *cache*: (tamanho de linha, associatividade, número de conjuntos).

Aplicação	No. simulações na segunda iteração	*Cache* de matrizes particionado da primeira iteração				*Cache* de matrizes particionado da segunda iteração			
		$(b_1, n_1, m_1)\|$ (b_2, n_2, m_2)	$f_1\|f_2$	taxa de falta (x1000)	tempo (ns)	$(b_1, n_1, m_1)\|$ (b_2, n_2, m_2)	$f_1\|f_2$	taxa de falta (x1000)	tempo (ns)
Cache de 8K *bytes*									
FFT	1	(64, 1, 32)| (32, 3, 64)	9,52| 90,48	17,97	3,91	(32, 2, 64)| (16, 1, 128)	84,41| 15,59	23,30	3,98
Cache de 12K *bytes*									
Conv.	1	(64, 2, 64)| (32, 1, 128)	67,99| 32,01	25,11	5,82	(64, 1, 64)| (32, 1, 128)	64,7| 35,3	58,87	8,73
FFT	1	(64, 1, 64)| (32, 2, 128)	9,52| 90,48	5,78	2,14	(32, 1, 128)| (16, 1, 256)	84,41| 15,59	57,19	7,66
G3fax	1	(32, 1, 64)| (16, 5, 128)	76,43| 23,57	4,31	1,33	(16, 4, 128)| (8, 1, 256)	9,95| 90,05	45,40	4,48
JPEG	1	(64, 1, 64)| (32, 2, 128)	90,73| 9,27	1,85	1,07	(32, 1, 128)| (16, 1, 256)	21,54| 78,46	21,00	3,35
MPEG	7	(64, 2, 64)| (32, 1, 128)	20,21| 78,79	1,88	1,22	(64, 1, 64)| (32, 1, 128)	3,74| 96,26	7,88	1,82

Maiores detalhes sobre as frações de alternâncias no acesso às metades das linhas e os resultados das simulações iterativas podem ser encontrados nos Apêndices V e VII, respectivamente.

5.7 Comparação com Trabalhos Anteriores

Nesta seção realizamos uma comparação qualitativa e quantitativa do nosso trabalho com os trabalhos anteriores.

5.7.1 Comparação Qualitativa

Da mesma forma que o nosso algoritmo, Lee, Park e Kim [19], Sánchez, González e Valero [16], González, Aliagas e Valero [15] e Naz, Kavi, Rezaei e Li [17], [18] mapeiam as variáveis de uma aplicação embarcada em *caches* particionados de acordo com a sua localidade.

A organização de *cache* proposta pelo nosso algoritmo pode ser considerada uma extensão da metodologia de Naz, Kavi, Rezaei e Li [17], [18], que mapeia variáveis escalares e matriciais a diferentes partições de *cache*. Neste sentido, nossa abordagem poderia ser vista como um particionamento adicional do *cache* de matrizes, de acordo com uma análise baseada em rastro de acesso as linhas do *cache*. Por outro lado, como nenhum *buffer* é considerado em nossos *caches* particionados, pode ainda haver espaço para mais melhorias no desempenho e consumo de energia.

Ao contrário do nosso algoritmo, Panda, Dutt e Nicolau [20] e Sánchez, González e Valero [16] determinam a configuração de memória ótima através da análise de código, eliminando a simulação de rastros de acesso à memória.

Para aumentar ainda mais os ganhos de desempenho e consumo de energia, nosso algoritmo poderia empregar a metodologia de Zhang e Vahid [4] para destinar parte do espaço de memória para um *cache* de vítimas configurável.

Ao contrário da metodologia de Givargis, Henkel e Vahid [14], o escopo do nosso algoritmo não incluiu a configuração de barramento de *cache*, visto que nosso foco está na configuração dos parâmetros de *cache*.

A metodologia proposta no nosso algoritmo para projetar *caches* de matrizes nível 1 particionados baseada em simulações de rastros de acesso à memória difere das metodologias anteriores de três maneiras.

Primeiro, começamos com uma organização de *cache* associativo por conjunto e uma restrição no tamanho de *cache*, de modo que o tamanho de *cache* não constitui um parâmetro configurável. Em segundo lugar, nosso método de particionamento pode ser aplicado iterativamente para definir

múltiplos *caches* nível 1, cada um deles ajustado para o conjunto de matrizes mapeadas. Por fim, nosso critério de seleção do melhor *cache* particionado é baseado no tempo médio de acesso à memória, sendo o consumo de energia e taxa de faltas também avaliados. Se considerarmos os acessos paralelos a dados gerados pelos processadores superescalares ou *multicores*, nossos *caches* de matrizes particionados baseados em rastros de acesso à memória podem melhorar ainda mais o desempenho, por permitirem acessos paralelos às partições de *caches*.

5.7.2 Comparação Quantitativa

Nesta seção, comparamos nossos resultados com os obtidos por trabalhos anteriores para *caches* associativos por conjunto. Para realizar esta comparação, não chegamos a executar os algoritmos dos trabalhos anteriores, mas nos baseamos nos resultados publicados.

Mesmo quando um trabalho utiliza simulação baseada em rastro de acesso à memória, não é possível realizar uma comparação precisa, devido às diferenças em relação às abordagens utilizadas por nosso trabalho e os trabalhos anteriores em relação ao tamanho e configuração de *caches*, modelos de consumo de energia, códigos fonte de aplicações e entradas utilizadas para exercitar as aplicações. Alguns trabalhos utilizaram potência como modelo de consumo de energia, por exemplo. Mesmo trabalhos que usam o mesmo modelo de desempenho e consumo de energia que o nosso podem não oferecer uma comparação justa se a tecnologia considerada para obter os valores no CACTI [32] for diferente da nossa, por exemplo.

A Tabela 5.12 compara os nossos resultados com os de trabalhos anteriores apresentando a configuração de memória, variação na percentagem de melhoria no desempenho, consumo de energia e taxa de falta para as aplicações consideradas.

A técnica apresentada por Lee, Park e Kim [19] adiciona um *buffer* completamente associativo com tamanho de linha grande a um *cache* diretamente mapeado com tamanho de linha pequeno e combina-os a uma unidade de *hardware* de *prefetch*. Embora [19] tenha obtido uma melhoria no consumo de energia maior que a do nosso trabalho, é possível que nossos resultados sejam comparáveis aos de [19], pois [19] utiliza 12,5% mais memória que nossos *caches* de matrizes particionados, correspondente a um *buffer* espacial de 1K *bytes*, e a diferença entre as máximas percentagens de melhoria no consumo de energia obtido por [19] e o nosso trabalho é de somente 10%.

Tabela 5.12 Comparação com trabalhos anteriores.

Trabalho	Configuração de memória	Variação de Melhoria na Taxa de Falta	Variação de Melhoria no Tempo Médio de Acesso à Memória	Variação de Melhoria no Consumo de Energia
Nosso Trabalho	*cache* de 8K *bytes* de matrizes, nível 1	-84,58% a 0,09%	-17,55% a 26,85%	-37,77% a 50,80%
Nosso Trabalho	*cache* de 12K *bytes* de matrizes, nível 1	-53,76% a 29,94%	2,61% a 30,33%	-6,34 a 33,67%
[19]	*cache* de 8K *bytes* dados e *buffer* espacial de 1K *bytes*.	-	-	10 a 60% (*)(****)
[16]	*cache* de dados de 8K *bytes* e *buffer* de até 256 *bytes*	~(-13%) a ~29% (**)	~(-3,19)% a ~10,52% (**)	-
[12]	*cache* de 32K *bytes* de dados, nível 1	5,11% a 41,18% (***)	-	-2,89% a 11,40% (*)(***)

- inexistência de dados para comparação.
(*) a métrica é consumo de potência.
(**) a comparação é com um *cache* diretamente mapeado de 8K *bytes* e linhas de 32 *bytes* não otimizado para as aplicações.
(***) a comparação é com um *cache* associativo por conjunto de 32K *bytes* com 4 vias.
(****) a comparação é com as seguintes configurações de *cache*:

1. *cache* de vítimas de 8K *bytes* e *buffer* de vítimas de 1K *bytes*;
2. *cache* diretamente mapeado de 16K *bytes*;
3. *cache* diretamente mapeado de 32K *bytes*;
4. *cache* diretamente mapeado de 64K *bytes*;
5. *cache* associativo por conjunto de 16K *bytes* com associatividade 2;
6. *cache* associativo por conjunto de 16K *bytes* com associatividade 4;
7. *cache* associativo por conjunto de 32K *bytes* com associatividade 2.

O método de projeto proposto por Sánchez *et al* [16] classifica, em tempo de compilação, as estruturas em espaciais, temporais ou não-armazenáveis em *cache*. As estruturas de dados espaciais e temporais são mapeadas em diferentes partições de um *cache* dual, ao passo que as não-armazenáveis em *cache* nunca são trazidas a este *cache*.

A metodologia de [16] usa 6,25% mais memória que nossos *caches*, referente à memória da partição temporal de 16 *words* do *cache* dual. Esta metodologia consegue uma expressiva redução na taxa de falta, quando comparada com a nossa metodologia, mas não consegue valores máximos de melhoria no tempo médio de acesso à memória maiores que os nossos.

A fim de reduzir a taxa de faltas, Bornoutian e Orailoglu [12] propõem uma arquitetura capaz de reconfigurar dinamicamente o *cache* para evitar poluição e interferência. A redução na taxa de falta encontrada em [12] é consideravelmente maior que a obtida por nosso trabalho, pois o objetivo principal de [12] é reduzir a taxa de falta, enquanto o nosso maior objetivo é reduzir o tempo médio de acesso à memória. Em contrapartida, nossos valores máximos de melhoria no consumo de energia são bem maiores que os de [12], apesar dos *caches* comparados serem de tamanhos diferentes.

Vale notar que, ao contrário do nosso trabalho, os percentuais de melhoria de [12] e [16] são comparados a um *cache* convencional de mesmo tamanho, não necessariamente ajustado para otimizar

os resultados, o que pode ter levado a resultados melhores que se eles tivessem sido comparados a *caches* convencionais ajustados para otimizar os resultados.

5.8 Conclusão

Neste capítulo apresentamos os resultados das simulações do nosso algoritmo de geração de *caches* de matrizes particionados, obtidos através de 310 simulações de 5 aplicações embarcadas para *caches* de 8 e 12K *bytes*.

A tempo das simulações aumentava com o tamanho do rastro e o tamanho do *cache* de matrizes unificado, variando de 20 minutos a 1 dia.

As aplicações Convolução, Fourier e G3fax demonstraram a utilização do algoritmo em aplicações com acesso a um número reduzido de matrizes. Por sua vez, as aplicações JPEG e MPEG comprovaram a eficácia do algoritmo em aplicações com maior número de matrizes acessadas e padrões de acesso irregulares.

A comparação de nossos resultados com o de trabalhos anteriores mostra que as melhorias medidas em nossos experimentos são da mesma ordem que as reportadas por outros métodos de configuração de *caches*.

No próximo capítulo descrevemos as contribuições deste trabalho e sugestões de desenvolvimentos para aperfeiçoamento de nosso algoritmo.

Capítulo 6

Conclusão

6.1 Introdução

Neste capítulo descrevemos nossas contribuições e sugestões de trabalhos futuros para aprimoramento do nosso algoritmo.

6.2 Contribuições

Neste trabalho, propomos uma metodologia de projeto de *cache* de matrizes, nível 1, particionados baseados na análise da simulação do rastro de acesso às matrizes de aplicações embarcadas. Os resultados de nossas simulações revelaram que a configuração dos parâmetros e o mapeamento de matrizes aos *caches* de matrizes particionados são capazes de melhorar o tempo médio de acesso à memória e consumo de energia, sem necessitar de aumentos no tamanho do *cache*. Estes resultados foram obtidos para aplicações com números distintos de matrizes.

Desenvolvemos um procedimento para instrumentação manual do código fonte de aplicações para a geração dos rastros de acesso à memória. Para implementar o algoritmo, construímos um simulador de *cache*.

Para validar a eficiência da organização e mapeamento de matrizes nos *caches* propostos produzidos, reunimos um conjunto de aplicações de uso comum em sistemas embarcados.

6.3 Trabalhos Futuros

Entre nossos objetivos futuros encontram-se a inclusão de nosso algoritmo em compiladores e *linkers* e o desenvolvimento de algoritmos para a reconfiguração dinâmica de *cache* ativada a partir de partes de programas ou tipos de entradas.

Nosso algoritmo pode ser acoplado a um algoritmo de otimização de código que leve em conta o tamanho do *cache*.

O algoritmo pode também ser estendido para ser aplicado em *caches* de instruções, *caches* com vários níveis e *caches* de escalares. Para tanto, o gerador de rastros de acesso à memória deveria ser modificado para distinguir instruções, matrizes e escalares acessados e nosso simulador de *cache* precisaria simular vários níveis de *caches*.

O modelo de medida de desempenho adotado no nosso trabalho pode ser aprimorado através da adição dos efeitos dos barramentos relacionados ao *cache* e de estimativas da fração dos acessos paralelos a partir do paralelismo no nível de tarefa e de instrução. Novos critérios para a seleção de *caches* particionados podem também ser experimentados, como, por exemplo, a área.

Em relação à parte experimental deste trabalho, uma sugestão é simular rastros de acesso à memória de um número maior de aplicações embarcadas, que, de preferência, acessem um grande número de matrizes. É interessante também realizar simulações com *caches* de 16K *bytes*, para possibilitar comparações com a maioria dos trabalhos anteriores, que utiliza este tamanho de *cache*.

Para facilitar o processo de instrumentação do código, a tabela de símbolos do programa poderia ser consultada ou a ferramenta ATOM [37] de geração de rastros de acesso à memória poderia ser empregada. Outra alternativa seria substituir o simulador de *cache* desenvolvido neste trabalho pelo simulador de *cache* DINERO IV [38]. Desenvolvemos um simulador de *cache* para podermos usar o algoritmo junto com o simulador.

Os modelos podem ser refinados para distinguir acessos de leitura e escrita. Neste caso, o rastro também precisaria ser modificado.

Modelos detalhados de *cache* podem ser utilizados para avaliar o tempo de acesso e consumo de energia com mais precisão.

Referências Bibliográficas

[1] A. Burns, A. Weelings, *Real-Time Systems and Programming Languages*, Addison-Wesley, 1997.

[2] R. Lupers, *Code Optimization Techniques for Embedded Processors, Methods, Algorithms, and Tools*, Kluwer Academic Publishers, 2002.

[3] Fax Group, Wikipedia, http://en.wikipedia.org/wiki/Fax#Group, Fevereiro de 2010.

[4] C. Zhang, F. Vahid, "Using a Victim Buffer in an Application-Specific Memory Hierarchy", *Proceedings of the Design Automation and Test in Europe Conference*, February 2004.

[5] V. Suhendra, T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores"**,** *Annual ACM IEEE Design Automation Conference, Proceedings of the 45th annual Design Automation Conference*, June 2008.

[6] A. Gordon-Ross, F. Vahid, N. D. Dutt, "Automatic Tuning of Two-Level Caches to Embedded Applications", *Proceedings of the Design, Automation and Test in Europe Conference*, 2004.

[7] D.A. Patterson, J.L. Hennessy, *Computer Architecture: A Quantitative Approach, Morgan Kaufmann*, Fourth Edition, 2007.

[8] A. Gordon-Ross, S. Cotterell, F. Vahid, "Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example"*, IEEE Computer Architecture Letters, Volume 1, Issue 1*, January 2002.

[9] A. Dominguez, S. Udayakumaran, R. Barua, "Heap Data Allocation to Scratch-Pad Memory in Embedded Systems", University of Maryland at College Park, January 2007.

[10] L. H. Lee, B. Moyer, J. Arends, "Instruction Fetch Energy Reduction Using Loop Caches For Embedded Applications with Small Tight Loops", *Proceedings of the 1999 International Symposium on Low Power Electronics and Design***,** August 1999.

[11] S. Cotterell, F. Vahid, "Synthesis of Customized Loop Caches for Core-Based Embedded Systems", *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, November 2002.

[12] G. Bournoutian, A. Orailoglu, "Miss Reduction in Embedded Processors Through Dynamic, Power-Friendly Cache Design ", *Proceedings of the Design Automation Conferenc*e, June 2008.

[13] A. Gordon-Ross, F. Vahid, N. D. Dutt, “Fast Configurable-Cache Tuning With a Unified Second-Level Cache”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 17, No. 1, January 2009.

[14] T. Givargis, J. Henkel, F. Vahid, “Interface and Cache Power Exploration for Core-Based Embedded System Design”, *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, San Jose, November 1999.

[15] A. González, C. Aliagas, M. Valero, “A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality”, *Proceedings of the International Conference of Supercomputing*, July 1995.

[16] F. J. Sánchez, A. González, M. Valero, “Software Management of Selective and Dual Data Caches”, *IEEE TCCA NEWSLETTERS*, pp.3 -10, March 1997.

[17] A. Naz, K. Krisha, M. Rezael, W. Li, “Making a Case For Split Data Caches For Embedded Applications”, *ACM SIGARCH Computer Architecture News*, Vol. 34, No.1, March 2006.

[18] A. Naz, K. Kavi, P. Sweany, M. Rezaei, “A Study of Separate Array and Scalar Caches", *Proceedings of the 18th International Symposium on High Performance Computing Systems and Applications (HPCS 2004)*, Manitoba, Canada, May 16-19, 2004.

[19] J-H. Lee, G-H. Park, S-D. Kim, “Dual Cache Architecture for Low Cost and High Performance”, *ETRI Journal*, October 2003.

[20] P. R. Panda, N. D. Dutt, A. Nicolau, “Architectural Exploration and Optimizations of Local Memory in Embedded Systems”, *Proceedings of the 10th International Symposium on System Synthesis*, 1997.

[21] M. Wolf, M. Lam, "A Data Locality Optimizing Algorithm", *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991.

[22] Tecnologias MIPS, www.mips.com, Setembro 2009.

[23] ARM – A Arquitetura Para O Mundo Digital, www.arm.com, Setembro 2009.

[24] Virage Logic, www.arccores.com, Setembro 2009.

[25] Tensilica, Tensilica : Núcleos de Processadores Customizáveis para os Planos de Dados, www.tensilica.com, Setembro 2009.

[26] Corporação Altera, Desenvolvimento do Sistema de Processador Embarcado Nios, www.altera.com/corporate/news_room/releases/products/nr-nios_delivers_goods.html, Setembro 2009.

[27] I. Issenin, E. Brockmeyer, M. Miranda, N. Dutt, "Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, 2004.

[28] Aplicação convolução, http://inside.mines.edu/~dhale/jtk/bench/DspBench.cpp, Outubro 2008.

[29] Aplicação transformada de Fourier, http://faculty.prairiestate.edu/skifowit/fft, Dezembro de 2008.

[30] Aplicação câmera digital, http://esd.cs.ucr.edu/digcam/DIGCAM/SYSLEVEL/cntrl.c, Agosto de 2008.

[31] MPEG.ORG, Aplicação decodificador MPEG-2, http://www.mpeg.org/MPEG/video/mssg-free-mpeg-software.html, Janeiro 2009.

[32] CACTI 5.3 (revisão 174): http://quid.hpl.hp.com:9081/cacti/, Agosto de 2009.

[33] Corporação Samsung, Guia de Produto DDR SDRAM 266, Versão Dezembro 2007, http://www.samsung.com/global/business/semiconductor/products/dram/downloads/ddr_product_guide_dec_07.pdf, Agosto 2009.

[34] C. Zhang, "An Efficient Direct Mapped Instruction Cache for Application-Specific Embedded Systems", *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 19-21 Setembro 2005.

[35] Diferença entre Variáveis Globais e Estáticas, http://c.ittoolbox.com/documents/difference-between-static-global-variable-12174, Janeiro de 2010.

[36] Convolução, Wikipédia, http://en.wikipedia.org/wiki/Convolution, Setembro de 2009.

[37] A. Eustance, A. Srivastava, "ATOM: A Flexible Interface for Building High Performance Analysis Tools", Western Research Laboratory, TN-44, 1994.

[38] DINERO IV Simulador de *Cache* Uniprocessador Dirigido por Rastro, http://pages.cs.wisc.edu/~markhill/DineroIV/, Outubro de 2009.

[39] C. Zhang, F. Vahid, R. Lysecky, "A Self-Tuning Cache Architecture for Embedded Systems", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, pp. 1-19, Maio 2004.

[40] A. Ghosh, T. Givargis, "Analytical Design Space Exploration of Caches for Embedded Systems", *Proceedings of the conference on Design, Automation and Test in Europe*, vol. 1, 2003.

[41] F. Vahid, T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, John Wiley & Sons, Inc., 2002.

[42] Modelos de *Benchmark*, Powerstone, http://www.cprover.org/software/benchmarks/, Janeiro de 2010.

[43] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, P. Manwedel, "Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems", *Proceedings of the tenth International Symposium on Hardware/Software Codesign*, May, 2002.

Apêndice I

Simulador de *Cache*

Neste apêndice descrevemos o simulador de *cache* desenvolvido neste trabalho para ser utilizado em conjunto com o nosso algoritmo de particionamento de *caches,* nível 1, de matrizes. Nosso simulador de *cache* foi projetado para *caches* associativos por conjunto que operam de acordo com a política de substituição do bloco menos recentemente usado (LRU).

I.1 Simulador de *Cache*

I.1.1 Estrutura de Dados

O *cache* associativo por conjunto de *n* vias com *m* conjuntos que seguem a política de substituição do bloco menos recentemente usado (LRU) é representado em nosso simulador pela estrutura de dados *CACHE* ilustrada na Figura I.1. Esta estrutura consiste num vetor de *m* elementos, onde cada elemento contém uma lista ligada *LISTA_DE_VIAS* de *n* nós e um vetor com *n* linhas de *cache*.

A *LISTA_DE_VIAS* implementa a política de substituição de blocos do bloco menos recentemente usado (LRU) indicando a ordem das vias acessadas no tempo.

A Figura I.2 mostra que cada linha do *cache* armazena as seguintes informações:

- a etiqueta dos dados atualmente armazenados na linha *L*;
- a última metade *Última* da linha *L* acessada;
- o número de alternâncias *Alternâncias* nos acessos às metades da linha *L*;
- o número de acessos *Acessos* à linha *L*.

Cada linha *L* de *CACHE* é identificada unicamente pela combinação de seu conjunto de linha de *cache Conjunto* e via *v*.

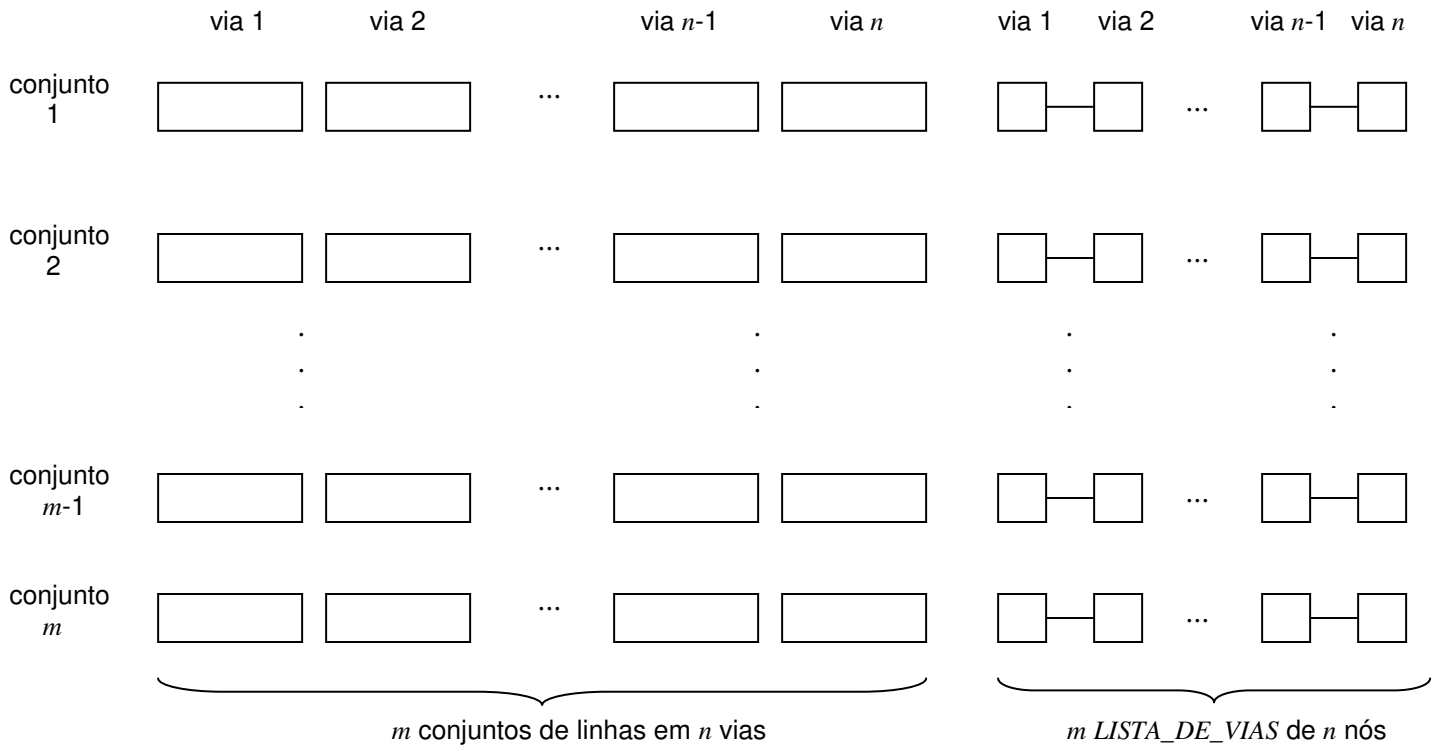


Figura I.1 Estrutura de dados *CACHE*.

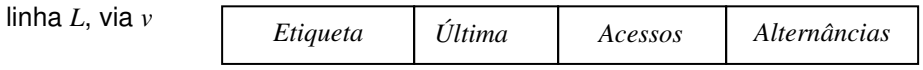


Figura I.2 Informações armazenadas em cada linha do *CACHE*.

Quando a referência *Ref* sai de *CACHE*, se não armazenarmos em alguma outra estrutura de dados o número de *Acessos* e *Alternâncias* ocorridos durante sua permanência no *cache*, esta informação é perdida. Por isso, armazenamos estas informações na estrutura de dados *LISTA_GLOBAL* descrita na Figura I.3 as informações de todos os N_t blocos de memória distintos que foram trazidos alguma vez ao *CACHE*. Enquanto a estrutura de dados *CACHE* armazena os dados atualmente no *cache*, a estrutura de dados *LISTA_GLOBAL* armazena todos os dados que foram

referenciados na aplicação. Portanto, *LISTA_GLOBAL* contém inclusive dados armazenados no *CACHE*.

Cada elemento em *LISTA_GLOBAL* é identificado unicamente pelo par *Conjunto* e *Etiqueta*.

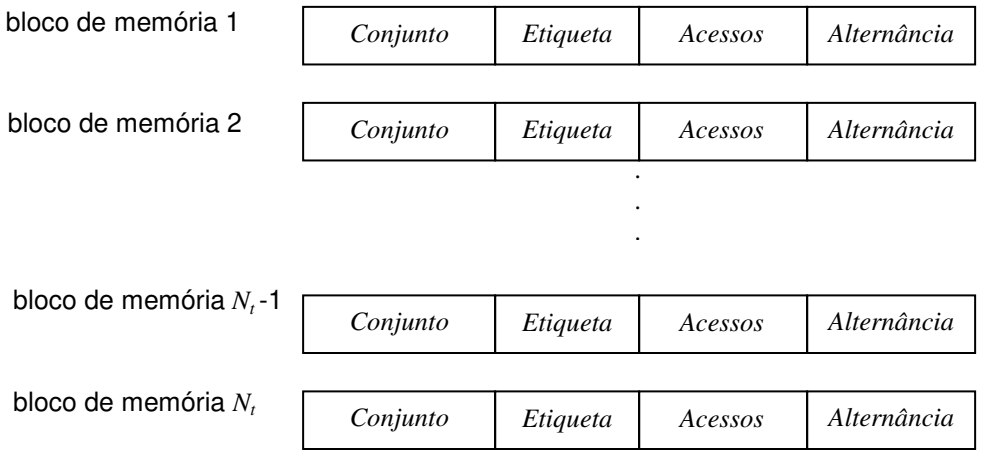


Figura I.3 Estrutura de dados *LISTA_GLOBAL*.

I.1.2 Operação

A Figura I.4 resume o funcionamento do simulador de *cache*.

Primeiramente, inicializamos as informações armazenadas em *CACHE*, nas linhas 1 a 5, e em *LISTA_GLOBAL*, nas linhas 6 a 8.

Para toda referência *Ref* do rastro de acesso à memória, calculamos o conjunto de linha de *cache* *Conjunto*(*Ref*) para o qual ela é mapeada e sua etiqueta *Etiqueta*(*Ref*), nas linhas 10 e 11, segundo cálculos especificados na seção 2.3.2 do Capítulo 2. Com base nestes dois valores, procuramos pelo seu respectivo elemento em *LISTA_GLOBAL* na linha 14. Caso nenhum elemento seja encontrado, trata-se do primeiro acesso a um novo bloco de memória, que é então inserido como um novo elemento na *LISTA_GLOBAL* na linha 16. Como mais um bloco de memória distinto foi acessado, o número de blocos distintos acessados N_t é incrementado na linha 17.

Das linhas 18 a 21, o simulador procura em cada uma das vias *v* a linha que contém a mesma etiqueta que a etiqueta da referência *Ref* *Etiqueta*(*Ref*). Caso alguma linha contenha esta mesma etiqueta, a variável *acertoCache* torna-se verdadeira, indicando que houve um acerto no *CACHE*.

Quando há falta no *CACHE*, o simulador soma o número de *Acessos* e *Alternâncias* do bloco substituído aos valores armazenados na *LISTA_GLOBAL* nas linhas 23 a 26. Esta atualização da *LISTA_GLOBAL* só é feita quando o bloco é substituído para evitar tempo de processamento do simulador em atualizações desnecessárias. Além disso, em caso de falta, na linha 27 o simulador escolhe a via *v'* localizada no início da *LISTA_ DE_VIAS* para ser substituída pelo novo bloco.

```
1  para (v=1 a n e Conjunto=1 a m)
2  |  CACHE[Conjunto, v].Alternâncias = 0;
3  |  CACHE[Conjunto, v].Acessos = 0;
4  |  CACHE[Conjunto, v].Última = Primeira Metade da Linha;
5  |  inicialize CACHE[Conjunto, v].LISTA_ DE_VIAS segundo a ordem 1 a n;

6  para (i=1 a Número_Máximo_Elementos_Distintos_Acessados_Suportado)
7  |  LISTA_GLOBAL[i].Alternâncias = 0;
8  |  LISTA_GLOBAL[i].Acessos = 0;

9  enquanto houver Ref não lida no rastro de acesso à memória
10 |  calcule Conjunto(Ref);
11 |  calcule Etiqueta(Ref);
12 |  acertoCache = falso;
13 |  v = 1;

14 |  procure na LISTA_GLOBAL elemento com valores Conjunto(Ref) e Etiqueta(Ref);

15 |  se (elemento não encontrado)
16 |  |  insira na LISTA_GLOBAL elemento com Conjunto(Ref) e Etiqueta(Ref);
17 |  |  N_i++;

18 |  enquanto ( (!acertoCache) && (v ≤ n))
19 |  |  se (CACHE[Conjunto(Ref), v].Etiqueta == Etiqueta(Ref))
20 |  |  |  acertoCache = verdadeiro;
21 |  |  v++;

22 |  se (!acertoCache)
23 |  |  LISTA_GLOBAL[Conjunto(Ref),  Etiqueta(Ref)].Alternâncias +=
24 |  |  CACHE[Conjunto(Ref), v].Alternâncias;
25 |  |  LISTA_GLOBAL[Conjunto(Ref),  Etiqueta(Ref)].Acessos +=
26 |  |  CACHE[Conjunto(Ref), v].Acessos;
27 |  |  v = CACHE[Conjunto(Ref)].LISTA_ DE_VIAS.Ponteiro_Início;
28 |  |  CACHE[Conjunto(Ref), v].Etiqueta = Etiqueta(Ref);

29 |  CACHE[Conjunto(Ref)].LISTA_ DE_VIAS.Ponteiro_Fim = v;
30 |  atualize CACHE[Conjunto(Ref), v].Última;
31 |  atualize CACHE[Conjunto(Ref), v].Alternâncias;
32 |  CACHE[Conjunto(Ref), v].Acessos++;

33 para (v=1 a n e Conjunto=1 a m)
34 |  copie CACHE[Conjunto, v].Alternâncias para LISTA_GLOBAL;
35 |  copie CACHE[Conjunto, v].Acessos para LISTA_GLOBAL;
```

Figura I.4 Procedimento da fase de *Simulação de Cache I.*

Em seguida, na linha 29 a via recém acessada *v* vai para o final da lista ligada *LISTA_ DE_VIAS*, de modo que o ponteiro para o final da lista sempre contém o nó da via que foi mais recentemente usada, enquanto o ponteiro para o início da lista aponta para o nó com a via menos recentemente usada.

Tanto em caso de acerto quanto de falta no *CACHE*, os valores de *Última* são atualizados e os valores de *Acessos* e *Alternâncias* são somados aos valores já existentes para o bloco em questão nas linhas 30 a 32.

Depois que todas as referências do rastro de acesso à memória são consideradas, o simulador copia nas linhas 33 a 35 o número de acessos e alternâncias de todos os blocos ainda presentes no *CACHE* para a *LISTA_GLOBAL*, pois, como este procedimento só é feito no momento de substituição do bloco de *cache*, precisamos fazer esta última cópia quando todos os acessos ao *cache* terminam.

Apêndice II

Tamanho e Espaço Adicional de Memória das Matrizes

Este apêndice mostra, para cada matriz acessada em nossas simulações, o seu tamanho e espaço de memória adicional devido à inserção de espaços para evitar a existência de múltiplas matrizes na mesma linha de *cache*.

II.1 Tamanho e Espaço Adicional de Memória

Analisando as Tabelas II.1 a II.5, vemos que, conforme esperado, a quantidade de espaço adicional diminui com a redução do tamanho de linha. Além disso, a fração de espaço adicional aumenta com a diminuição do tamanho da matriz. O espaço adicional é calculado pela Equação 5.1 do Capítulo 5.

Tabela II.1 Tamanho e espaço adicional referente às matrizes da aplicação Convolução.

A	Tamanho de *A* (*bytes*)	Espaço Adicional Referente a *A* (*bytes*)		
		b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
x	32	32	0	0
z	14000	16	16	0
y	13972	44	12	12

Tabela II.2 Tamanho e espaço adicional referente às matrizes da aplicação FFT.

A	Tamanho de *A* (*bytes*)	Espaço Adicional Referente a *A* (*bytes*)		
		b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
a	65544	56	24	8
ip	264	56	24	8
t	32776	56	24	8
w	81920	0	0	0

Tabela II.3 Tamanho e espaço adicional referente às matrizes da aplicação G3fax.

A	Tamanho de *A* (*bytes*)	Espaço Adicional Referente a *A* (*bytes*)		
		b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
fax	15264	32	0	0
g3black	18304	0	0	0
g3white	20512	32	0	0
rowbuf	5184	0	0	0

Tabela II.4 Tamanho e espaço adicional referente às matrizes da aplicação JPEG.

A	Tamanho de *A* (*bytes*)	Espaço Adicional Referente a *A* (*bytes*)		
		b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
buffer	4224	0	0	0
buffer2	4096	0	0	0
buffer3	8192	0	0	0
COS_TABLE	128	0	0	0
ibuffer	64	0	0	0
imageFileHandle	32	32	0	0
imageFileName	1	63	31	15
obuffer	128	0	0	0
QuantShiftTable	64	0	0	0
s	64	0	0	0

Tabela II.5 Tamanho e espaço adicional referente às matrizes da aplicação MPEG.

A	Tamanho de *A* (*bytes*)	Espaço Adicional Referente a *A* (*bytes*)		
		b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
auxframe	12	12	20	4
auxframe_0	16384	0	0	0
auxframe_1	4096	0	0	0
auxframe_2	4096	0	0	0
backward_reference_frame	12	12	20	4
backward_reference_frame_0	16384	0	0	0
backward_reference_frame_1	4096	0	0	0
backward_reference_frame_2	4096	0	0	0
base	4696	24	8	8
BMBtab0	32	32	0	0
BMBtab1	16	16	16	0
c	512	0	0	0
CBPtab0	64	0	0	0
Clip	1024	0	0	0
current_frame	12	12	20	4
dc_dct_pred	12	12	20	4
DCchromtab0	64	0	0	0
DCchromtab1	64	0	0	0
DClumtab0	64	0	0	0
DCTtab0	180	52	12	12
DCTtab0a	756	52	12	12
DCTtab1	24	24	8	8
DCTtab1a	24	24	8	8
DCTtab2	48	48	16	0
DCTtab3	48	48	16	0
DCTtab6	48	48	16	0
DCTtabfirst	36	36	28	12
DCTtabnext	36	36	28	12
default_intra_quantizer_matrix	64	0	0	0
dmvector	8	8	24	8
enhan	4696	24	8	8
f_code	4	4	28	12
forward_reference_frame	12	12	20	4
forward_reference_frame_0	16384	0	0	0
forward_reference_frame_1	4096	0	0	0
forward_reference_frame_2	4096	0	0	0
frame_rate_Table	128	0	0	0
motion_vertical_field_select	16	16	16	0
Non_Linear_quantizer_scale	32	32	0	0
obfr	4096	0	0	0
outname	32	32	0	0
PMBtab0	16	16	16	0
PMBtab1	16	16	16	0
PMV	32	32	0	0
scan	128	0	0	0
stwclass_table	9	9	23	7
Table_6_20	12	12	20	4
tmp	512	0	0	0
tmpname	32	32	0	0

Apêndice III

Parâmetros Usados na Avaliação de *Caches*

Neste Apêndice detalhamos os parâmetros usados na avaliação do *cache* descritos na seção 5.5.3, do Capítulo 5, para permitir a reprodução de nossos resultados.

III.1 Parâmetros Usados na Avaliação de *Caches*

Em nossos *caches*, consideramos que o barramento tem 32 *bits* de largura.

Os tempos de penalidade da falta foram obtidos da especificação de uma memória DRAM266 [33] da Samsung™, segundo a qual são necessários 67,5 ns para trazer a primeira *word* ao *cache* e 7,5 ns para trazer cada *word* adicional. Consideramos que o barramento tem 32 *bits* de largura. A Tabela III.1 mostra a penalidade da falta por tamanho de linha utilizado no nosso trabalho.

Tabela III.1 Penalidade da falta por tamanho de linha *b* [33].

b (*bytes*)	Penalidade da Falta (*ns*)
4	67,5
8	75
16	90
32	120
64	180

De acordo com [34], em microprocessadores e memórias comerciais típicas a penalidade de energia é de 50 a 200 vezes o valor da energia de acerto. Consideramos que, para cada tamanho de

cache e de linha, a penalidade de energia é 50 vezes a energia de acerto para um *cache* diretamente mapeado, para qualquer que seja a associatividade do *cache* avaliado.

Consideramos o valor 50 para evitar a supervalorização da redução do tamanho de linha na redução da penalidade de energia, possibilitando a avaliação do nosso algoritmo sob o valor de penalidade de energia menos favorável a melhoria de consumo de energia.

A Tabela III.2 mostra a penalidade de energia por tamanho de *cache* e tamanho de linha utilizado no nosso trabalho. As Tabelas III.3 e III.4 detalham os tempos e energia de acerto utilizados.

Tabela III.2 Penalidade de energia em nJ por tamanho de *cache c* e tamanho de linha *b*.

b (*bytes*)	*c* (K *bytes*)									
	1	2	3	4	5	6	7	8	10	12
64	6,32	4,15	4,34	4,19	4,29	4,38	4,48	4,65	4,84	5,04
32	4,27	2,81	2,91	1,84	1,94	2,04	4,84	2,26	2,45	2,64
16	1,81	1,80	1,87	1,79	1,88	1,31	2,07	2,29	2,43	1,93
8	0,85	0,65	1,08	1,09	1,41	1,13	1,28	1,61	1,76	1,92

Tabela III.3 Tempo de acerto em ns por tamanho de *cache* *c*, tamanho de linha *b* e associatividade *n*. Valores em negrito foram extraídos do CACTI. Demais valores foram interpolados conforme explicado no Capítulo 5.

b (*bytes*)	*n*	*c* (K *bytes*)									
		1	2	3	4	5	6	7	8	10	12
64	1	0,713	**0,715**	**0,745**	**0,730**	**0,746**	**0,759**	**0,772**	**0,806**	**0,829**	**0,851**
	2	1,135	1,155	1,196	**1,421**	**1,488**	**1,505**	**1,521**	**1,501**	**1,521**	**1,539**
	3	1,099	1,173	1,215	1,480	1,488	1,565	1,680	1,657	1,678	1,673
	4	1,063	1,192	1,234	1,539	1,488	1,626	1,838	**1,814**	**1,834**	**1,807**
	5	1,072	1,202	1,244	1,566	1,514	1,654	1,870	1,943	1,971	1,939
	6	1,081	1,213	1,255	1,593	1,540	1,682	1,901	2,072	2,107	2,072
	7	1,091	1,223	1,265	1,621	1,566	1,711	1,933	2,201	2,244	2,205
	8	1,100	1,234	1,276	1,648	1,592	1,739	1,964	2,331	2,380	2,338
32	1	**0,775**	**0,778**	**0,805**	**0,764**	**0,781**	**0,795**	**0,882**	**0,782**	**0,813**	**0,836**
	2	1,235	**1,256**	**1,294**	**1,277**	**1,298**	**1,316**	**1,333**	**1,308**	**1,330**	**1,349**
	3	1,195	1,276	1,347	1,330	1,351	1,369	1,386	1,346	1,366	1,384
	4	1,156	1,297	1,401	**1,383**	**1,404**	**1,422**	**1,439**	**1,384**	**1,403**	**1,420**
	5	1,166	1,308	1,413	1,408	1,429	1,447	1,541	1,483	1,508	1,524
	6	1,176	1,320	1,425	1,432	1,453	1,471	1,644	1,582	1,612	1,629
	7	1,186	1,331	1,437	1,457	1,478	1,496	1,747	1,680	1,716	1,733
	8	1,197	1,342	1,449	1,482	1,503	1,520	1,849	**1,779**	**1,821**	**1,837**
16	1	**0,742**	**0,783**	**0,811**	**0,838**	**0,864**	**0,804**	**0,868**	**0,890**	**0,903**	**0,841**
	2	**1,181**	**1,216**	**1,255**	**1,316**	**1,337**	**1,378**	**1,395**	**1,395**	**1,420**	**1,440**
	3	1,201	1,236	1,275	1,292	1,312	1,341	1,358	1,404	1,427	1,447
	4	1,220	**1,256**	**1,295**	**1,269**	**1,288**	**1,304**	**1,320**	**1,414**	**1,435**	**1,454**
	5	1,230	1,267	1,318	1,291	1,311	1,327	1,343	1,410	1,431	1,449
	6	1,241	1,278	1,341	1,314	1,333	1,350	1,365	1,406	1,427	1,444
	7	1,252	1,289	1,364	1,337	1,356	1,372	1,388	1,402	1,422	1,439
	8	1,263	1,300	1,387	**1,359**	**1,378**	**1,395**	**1,411**	**1,398**	**1,418**	**1,435**
8	1	**0,778**	**0,777**	**0,877**	**0,884**	**0,907**	**0,839**	**0,851**	**0,903**	**0,945**	**0,988**
	2	**1,283**	**1,310**	**1,349**	**1,292**	**1,280**	**1,301**	**1,328**	**1,387**	**1,420**	**1,401**
	3	1,242	1,262	1,299	1,292	1,298	1,305	1,328	1,383	1,411	1,412
	4	**1,201**	**1,214**	**1,249**	**1,292**	**1,316**	**1,310**	**1,328**	**1,378**	**1,402**	**1,423**
	5	1,211	1,224	1,260	1,293	1,316	1,316	1,333	1,385	1,410	1,431
	6	1,222	1,235	1,271	1,294	1,316	1,321	1,338	1,392	1,417	1,439
	7	1,232	1,246	1,281	1,295	1,316	1,326	1,343	1,399	1,425	1,447
	8	1,243	**1,256**	**1,292**	**1,296**	**1,315**	**1,332**	**1,348**	**1,406**	**1,433**	**1,455**

Tabela III.4 Energia consumida no acerto em nJ por tamanho de *cache* *c*, tamanho de linha *b* e associatividade *n*. Valores em negrito foram extraídos do CACTI.
Demais valores foram interpolados conforme explicado no Capítulo 5.

b (*bytes*)	*n*	*c* (K *bytes*)									
		1	2	3	4	5	6	7	8	10	12
64	1	0,126	**0,083**	**0,087**	**0,084**	**0,086**	**0,088**	**0,090**	**0,093**	**0,097**	**0,101**
	2	0,076	0,075	0,079	**0,138**	**0,209**	**0,211**	**0,214**	**0,208**	**0,211**	**0,213**
	3	0,077	0,097	0,102	0,186	0,281	0,283	0,401	0,391	0,393	0,282
	4	0,078	0,120	0,126	0,233	0,353	0,355	0,588	**0,573**	**0,575**	**0,351**
	5	0,089	0,137	0,143	0,274	0,413	0,415	0,688	0,769	0,922	0,558
	6	0,100	0,154	0,161	0,314	0,474	0,476	0,787	0,965	1,269	0,765
	7	0,111	0,171	0,178	0,354	0,534	0,536	0,887	1,161	1,615	0,973
	8	0,123	0,188	0,196	0,395	0,595	0,597	0,986	1,356	1,962	1,180
32	1	**0,085**	**0,056**	**0,058**	**0,037**	**0,039**	**0,041**	**0,097**	**0,045**	**0,049**	**0,053**
	2	0,051	**0,051**	**0,053**	**0,052**	**0,053**	**0,054**	**0,056**	**0,060**	**0,062**	**0,065**
	3	0,052	0,066	0,071	0,070	0,071	0,073	0,074	0,075	0,077	0,079
	4	0,053	0,081	0,090	**0,088**	**0,090**	**0,091**	**0,093**	**0,091**	**0,092**	**0,094**
	5	0,060	0,093	0,102	0,103	0,105	0,107	0,125	0,122	0,148	0,150
	6	0,068	0,104	0,115	0,118	0,120	0,122	0,156	0,153	0,204	0,205
	7	0,075	0,116	0,128	0,134	0,136	0,138	0,188	0,184	0,259	0,261
	8	0,083	0,127	0,140	0,149	0,151	0,153	0,220	**0,215**	**0,315**	**0,316**
16	1	**0,036**	**0,036**	**0,037**	**0,036**	**0,038**	**0,026**	**0,041**	**0,046**	**0,049**	**0,039**
	2	**0,022**	**0,023**	**0,025**	**0,041**	**0,042**	**0,044**	**0,043**	**0,048**	**0,050**	**0,053**
	3	0,028	0,031	0,032	0,041	0,042	0,043	0,044	0,055	0,057	0,059
	4	0,035	**0,038**	**0,039**	**0,041**	**0,042**	**0,043**	**0,044**	**0,062**	**0,064**	**0,065**
	5	0,040	0,043	0,046	0,048	0,049	0,050	0,051	0,066	0,067	0,068
	6	0,045	0,048	0,053	0,055	0,056	0,057	0,058	0,069	0,070	0,072
	7	0,050	0,054	0,060	0,062	0,063	0,065	0,066	0,072	0,074	0,075
	8	0,055	0,059	0,067	**0,069**	**0,071**	**0,072**	**0,073**	**0,075**	**0,077**	**0,078**
8	1	**0,017**	**0,013**	**0,022**	**0,022**	**0,028**	**0,023**	**0,026**	**0,032**	**0,035**	**0,038**
	2	**0,019**	**0,024**	**0,025**	**0,027**	**0,022**	**0,024**	**0,027**	**0,044**	**0,049**	**0,039**
	3	0,019	0,023	0,024	0,030	0,029	0,028	0,028	0,044	0,048	0,045
	4	**0,019**	**0,022**	**0,024**	**0,034**	**0,036**	**0,033**	**0,030**	**0,043**	**0,047**	**0,051**
	5	0,022	0,026	0,027	0,036	0,037	0,035	0,034	0,048	0,052	0,055
	6	0,025	0,029	0,030	0,038	0,039	0,038	0,037	0,053	0,056	0,059
	7	0,027	0,032	0,033	0,040	0,041	0,041	0,041	0,057	0,060	0,063
	8	0,030	**0,035**	**0,037**	**0,041**	**0,042**	**0,043**	**0,044**	**0,062**	**0,065**	**0,068**

Apêndice IV

Resultados da *Simulação de Cache I*

Neste apêndice apresentamos os resultados da *Simulação de Cache I* para *caches* de matrizes unificadas do nosso algoritmo, descritos na seção 4.2.3 do Capítulo 4, referentes ao número total de referências acessadas, número de blocos distintos acessados e taxa de faltas.

IV.1 Número Total de Referências Acessadas, Número de Blocos Distintos Acessados e Taxa de Faltas da *Simulação de Cache I*

A Tabela IV.1 apresenta o total de referências acessadas $NREF_0$ nos rastros de cada aplicação simulada.

Tabela IV.1 Número total de referências acessadas nos rastros de cada aplicação simulada.

Aplicação	$NREF_0$
Convolução	43692
FFT	426356
G3fax	325518
JPEG	464972
MPEG	628807

Para cada aplicação e configuração de *cache* de matrizes unificado, a Tabela IV.2 mostra o número $N_t(C_0)$ de blocos distintos acessados, o número $NF(C_0)$ e a taxa de faltas. A quantidade de linhas disponíveis no *cache* também é apresentada.

Tabela IV.2 Configuração de *cache* de matrizes unificado C_0, número $N_t(C_0)$ de blocos distintos acessados, número $NF(C_0)$ e taxa de faltas por aplicação.

b	n	No. blocos distintos em C_0	Convolução			FFT			G3fax			JPEG			MPEG		
			N_t (C_0)	NF (C_0)	Taxa de Falta (C_0) (x1000)	N_t (C_0)	NF (C_0)	Taxa de Falta (C_0) (x1000)	N_t (C_0)	NF (C_0)	Taxa de Falta (C_0) (x1000)	N_t (C_0)	NF (C_0)	Taxa de Falta (C_0) (x1000)	N_t (C_0)	NF (C_0)	Taxa de Falta (C_0) (x1000)
c = 8K *bytes*																	
64	8	128	439	1098	25,13	266	4705	11,03	369	910	2,80	274	530	1,14	302	1549	2,47
	4			1098	25,13		4544	10,65		1347	4,14		467	1,00		1890	3,01
	2			1098	25,13		4292	10,06		3775	11,60		415	0,89		2191	3,49
32	8	256	876	2190	50,12	531	8012	18,79	642	1062	3,26	540	1022	2,19	542	1010	1,61
	4			2190	50,12		7819	18,33		1077	3,31		910	1,95		1452	2,31
	2			2190	50,12		7368	17,28		2764	8,49		740	1,59		1708	2,72
16	8	512	1751	4377	100,17	1061	11516	27,01	1179	1575	4,84	1066	1972	4,24	1034	1463	2,33
	4			4377	100,17		11308	26,52		1575	4,84		1748	3,75		1646	2,62
	2			4377	100,17		11130	26,10		1856	5,70		1430	3,07		2043	3,25
c = 12K *bytes*																	
64	6	192	439	1094	25,03	266	1125	2,63	369	758	2,33	274	286	0,61	302	978	1,56
	3			1088	24,90		1823	4,27		1197	3,68		287	0,61		1116	1,78
32	6	384	876	2182	49,94	531	2084	4,88	642	974	2,99	540	558	1,20	542	798	1,27
	3			2169	49,64		3011	7,06		996	3,06		558	1,20		853	1,36
16	6	768	1751	4360	99,78	1061	3517	8,24	1179	1466	4,50	1066	1092	2,34	1034	1260	2,01
	3			4334	99,19		4639	10,88		1491	4,58		1092	2,34		1312	2,09

Analisando as Tabelas IV.1 e IV.2, vemos que a aplicação Convolução é a que possui o acesso mais distribuído pelas linhas, pois, apesar de acessar aproximadamente somente 10% do número total de referências das outras aplicações, possui o maior número $N_t(C_0)$ de blocos distintos acessados. Logo, a aplicação Convolução possui a menor localidade espacial. Para as aplicações Convolução, FFT, G3fax e JPEG, o tamanho de linha em que incide a menor taxa de falta é a linha de 64 *bytes*. Para a aplicação MPEG, o tamanho de linha com a menor taxa de faltas é 32 *bytes*. A menor incidência de faltas nestes tamanhos de linhas pode contribuir para o menor tempo médio de acesso à memória observado nestes tamanhos de linhas em *caches* de 4 e 16K *bytes* [7], como vemos pelo tamanho de linhas dos *caches* de matrizes particionados de menor tempo médio de acesso à memória listados na Tabela 5.5 do Capítulo 5 (Convolução, FFT, JPEG para *cache* de 8 e 12 K *bytes* e MPEG para *cache* de 12 K *bytes*).

Em FFT encontramos a maior taxa de faltas de capacidade dentre todas as aplicações, pois um crescimento de 50% no tamanho do *cache* resulta num decréscimo de 57 a 73% na taxa de falta. Por outro lado, na aplicação Convolução, as faltas não são de capacidade, visto que o aumento no tamanho do *cache* manteve a taxa de faltas praticamente inalterada.

Para um mesmo tamanho de *cache* e tamanho de linha, a taxa de faltas por conflito tende a diminuir com o aumento do grau de associatividade [7]. Entretanto, observamos o comportamento

inverso nas aplicações Convolução, G3fax e JPEG para ambos os tamanhos de *cache*, e na aplicação FFT no *cache* de 8K *bytes*. Este comportamento ocorre devido à freqüência e ordem na qual as referências conflitantes são acessadas. Como o índice da linha depende do número de conjuntos de linhas disponíveis, e como o número de conjuntos de linhas disponíveis muda com o grau de associatividade, as referências conflitantes mudam com o grau de associatividade.

Uma referência T com alta freqüência de acesso ao longo do tempo no padrão de acesso da Figura IV.1 pode permanecer no *cache* mapeado diretamente por não disputar o mesmo conjunto com outras referências, como ilustrado na Figura IV.2. A Figura IV.3 mostra que esta mesma referência T no *cache* de associatividade 2 pode entrar e sair constantemente do *cache* para dar espaço às suas referências conflitantes, resultando em (n-1) faltas de T e n faltas referentes à A_i, com i=1 a n. Já no *cache* mapeado diretamente há somente 1 falta de T e n faltas referentes à A_i, com i=1 a n, se T não disputar a linha com as referências A_i, com i=1 a n.

$$TA_1\ A_1\ A_2\ TA_2\ A_2\ A_3\ ...TA_{n-1}\ A_{n-1}\ A_n$$

Figura IV.1 Exemplo de padrão de acesso à memória.

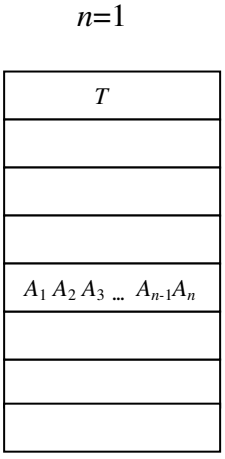
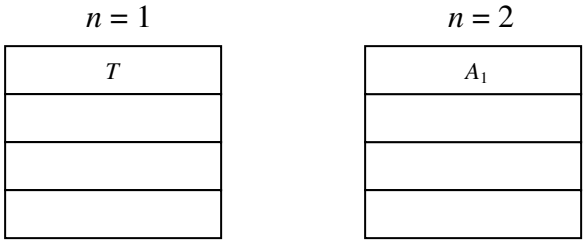
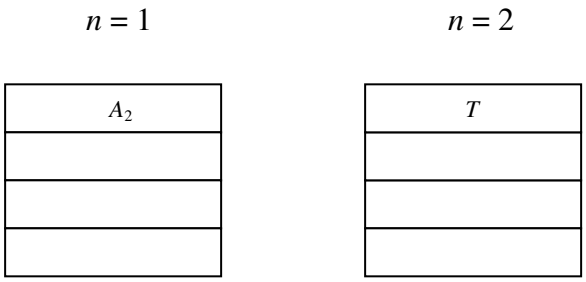


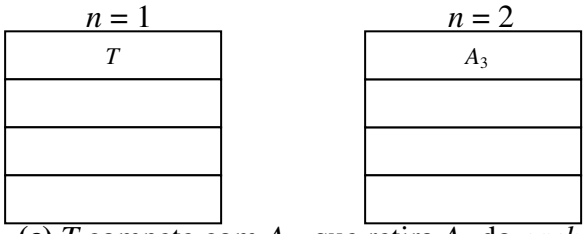
Figura IV.2 Acessos no *cache* mapeado diretamente.



(a) T gera 1 falta compulsória e A_1 gera 1 falta compulsória

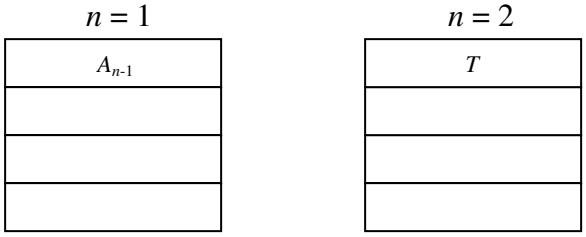


(b) A_2 compete com T, que retira A_1 do *cache*

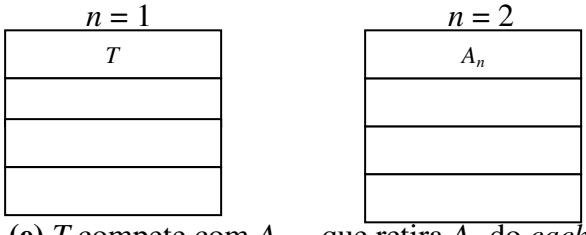


(c) T compete com A_2, que retira A_3 do *cache*

.
.
.



(d) $A_{n\text{-}1}$ compete com T, que retira $A_{n\text{-}2}$ do *cache*



(e) T compete com $A_{n\text{-}1}$, que retira A_n do *cache*

Figura IV.3 Acessos no *cache* com grau de associatividade 2.

Apêndice V

Frações de Alternância de Acessos e Números de Blocos Distintos Acessados para as Matrizes das Aplicações

Neste apêndice apresentamos os resultados da *Simulação de Cache I*, para *caches* de matrizes unificados, do nosso algoritmo, descritos na seção 4.2.3 do Capítulo 4, referentes à fração $M(A)$ de alternâncias de acessos às metades da linha e ao número $N(A)$ de blocos distintos acessados, para a matriz A.

V.1 $M(A)$ e $N(A)$ da Aplicação Convolução

As Tabelas V.1 e V.2 mostram a fração $M(A)$ de alternâncias de acesso às metades da linha da aplicação Convolução para *caches* de 8 e 12K *bytes*, respectivamente. A Tabela V.3 mostra o número $N(A)$ de blocos distintos acessados pela aplicação Convolução.

Tabela V.1 $M(A)$ das matrizes da aplicação Convolução para *cache* de 8K *bytes*.

A	$M(A)$								
	b=64 *bytes*			b=32 *bytes*			b=16 *bytes*		
	n=8	n=4	n=2	n=8	n=4	n=2	n=8	n=4	n=2
x	0,00	0,00	0,00	24,99	24,99	24,99	49,98	49,98	49,98
y	11,37	11,37	11,37	22,73	22,73	22,73	45,46	45,46	45,46
z	6,26	6,26	6,26	12,50	12,50	12,50	25,00	25,00	25,00

Tabela V.2 *M*(*A*) das matrizes da aplicação Convolução para *cache* de 12K *bytes*.

A	*M*(*A*)					
	b=64 *bytes*		*b*=32 *bytes*		*b*=16 *bytes*	
	n=6	*n*=3	*n*=6	*n*=3	*n*=6	*n*=3
x	0,00	0,00	12,50	12,50	49,98	49,98
y	11,37	11,37	24,99	24,99	45,46	45,46
z	6,26	6,26	22,73	22,73	25,00	25,00

Tabela V.3 *N*(*A*) das matrizes da aplicação Convolução.

A	*N*(*A*)		
	b=64 *bytes*	*b*=32 *bytes*	*b*=16 *bytes*
x	1	1	2
y	219	438	875
z	219	437	874

V.2 *M(A)* e *N(A)* da Aplicação FFT

As Tabelas V.4 e IV.5 mostram a fração *M*(*A*) de alternâncias de acesso às metades da linha da aplicação FFT para *caches* de 8 e 12K *bytes*, respectivamente. A Tabela IV.6 mostra o número *N*(*A*) de blocos distintos acessados pela aplicação FFT.

Tabela V.4 *M*(*A*) das matrizes da aplicação FFT para *cache* de 8K *bytes*.

A	*M*(*A*)								
	b=64 *bytes*			*b*=32 *bytes*			*b*=16 *bytes*		
	n=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2
A	6,42	6,37	6,32	14,62	14,65	14,63	41,40	41,40	41,79
Ip	44,37	44,38	44,37	27,47	27,47	27,47	29,92	29,49	29,49
T	5,10	5,10	4,87	11,46	7,64	11,46	38,12	38,12	37,80
W	9,15	9,56	9,30	7,51	8,34	8,53	10,01	11,54	11,54

Tabela V.5 *M*(*A*) das matrizes da aplicação FFT para *cache* de 12K *bytes*.

A	*M*(*A*)					
	b=64 *bytes*		*b*=32 *bytes*		*b*=16 *bytes*	
	n=6	*n*=3	*n*=6	*n*=3	*n*=6	*n*=3
a	6,67	6,59	15,35	15,19	42,06	42,06
ip	44,48	44,38	27,65	27,47	23,83	29,49
t	5,50	5,89	11,46	13,55	38,61	38,20
w	12,20	11,94	10,42	10,42	13,47	13,47

Tabela V.6 $N(A)$ das matrizes da aplicação FFT.

A	$N(A)$		
	b=64 *bytes*	b=32 *bytes*	b=16 *bytes*
a	128	256	512
ip	2	3	5
t	39	78	156
w	97	194	388

V.3 *M(A)* e *N(A)* da Aplicação G3fax

As Tabelas V.7 e V.8 mostram a fração $M(A)$ de alternâncias de acesso às metades da linha da aplicação G3fax para *caches* de 8 e 12K *bytes*, respectivamente. A Tabela V.9 mostra a quantidade $N(A)$ de blocos distintos acessados pela aplicação G3fax.

Tabela V.7 $M(A)$ das matrizes da aplicação G3fax para *cache* de 8K *bytes*.

A	$M(A)$								
	b=64 *bytes*			b=32 *bytes*			b=16 *bytes*		
	n=8	n=4	n=2	n=8	n=4	n=2	n=8	n=4	n=2
fax	6,26	6,26	6,26	12,51	12,51	12,51	25,01	25,01	25,01
g3black	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
g3white	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
rowbuf	12,02	12,02	12,02	24,06	24,06	23,95	48,35	48,35	48,28

Tabela V.8 $M(A)$ das matrizes da aplicação G3fax para *cache* de 12K *bytes*.

A	$M(A)$					
	b=64 *bytes*		b=32 *bytes*		b=16 *bytes*	
	n=6	n=3	n=6	n=3	n=6	n=3
fax	6,26	6,26	12,51	12,51	25,01	25,01
g3black	0,00	0,00	0,00	0,00	0,00	0,00
g3white	0,00	0,00	0,00	0,00	0,00	0,00
rowbuf	12,02	12,02	24,06	24,06	48,39	48,39

Tabela V.9 $N(A)$ das matrizes da aplicação G3fax.

A	$N(A)$		
	b=64 *bytes*	b=32 *bytes*	b=16 *bytes*
fax	240	478	955
g3black	86	91	93
g3white	15	18	23
rowbuf	28	55	108

V.4 *M*(*A*) e *N*(*A*) da Aplicação JPEG

As Tabelas V.10 e V.11 apresentam a fração *M*(*A*) de alternâncias de acesso às metades de linha da aplicação JPEG para os *caches* de 8 e 12K *bytes*, respectivamente, enquanto a Tabela V.12 mostra o número *N*(*A*) de blocos distintos acessados.

Tabela V.10 *M*(*A*) das matrizes da aplicação JPEG para *cache* de 8K *bytes*.

A	*M*(*A*)								
	b=64 *bytes*			*b*=32 *bytes*			*b*=16 *bytes*		
	n=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2
buffer	6,26	6,67	6,67	12,51	12,51	12,51	25,01	25,01	25,01
buffer2	8,34	8,34	10,42	16,67	16,67	20,84	33,34	33,34	41,67
buffer3	6,26	7,82	7,82	9,38	12,51	12,51	37,51	37,51	37,51
COS_TABLE	100,00	100,00	100,00	0,00	0,00	0,00	11,11	11,11	11,11
ibuffer	12,50	12,50	12,50	25,00	25,00	25,00	50,00	50,00	50,00
imageFileHandle	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
imageFileName	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
obuffer	12,46	12,46	12,46	24,91	24,91	24,91	49,81	49,81	49,81
QuantShiftTable	18,04	18,04	18,04	24,81	24,81	24,81	49,61	49,61	49,61
s	14,29	14,29	14,29	25,00	25,00	25,00	50,00	50,00	50,00

Tabela V.11 *M*(*A*) das matrizes da aplicação JPEG para *cache* de 12K *bytes*.

A	*M*(*A*)					
	b=64 *bytes*		*b*=32 *bytes*		*b*=16 *bytes*	
	n=6	*n*=3	*n*=6	*n*=3	*n*=6	*n*=3
buffer	6,67	6,67	12,51	12,51	25,01	25,01
buffer2	10,42	10,42	20,84	20,84	41,67	41,67
buffer3	7,82	7,82	15,63	15,63	43,76	43,76
COS_TABLE	100,00	100,00	0,00	0,00	11,11	11,11
ibuffer	12,50	12,50	25,00	25,00	50,00	50,00
imageFileHandle	0,00	0,00	0,00	0,00	0,00	0,00
imageFileName	0,00	0,00	0,00	0,00	0,00	0,00
obuffer	12,46	12,46	24,91	24,91	49,81	49,81
QuantShiftTable	18,04	18,04	24,81	24,81	49,61	49,61
s	14,29	14,29	25,00	25,00	50,00	50,00

Tabela V.12 $N(A)$ das matrizes da aplicação JPEG.

A	$N(A)$		
	b=64 *bytes*	b=32 *bytes*	b=16 *bytes*
buffer	66	132	264
buffer2	65	129	257
buffer3	129	257	512
COS_TABLE	2	4	4
ibuffer	3	5	8
imageFileHandle	1	1	1
imageFileName	1	1	1
obuffer	3	5	9
QuantShiftTable	2	3	5
s	2	3	5

V.5 *M*(*A*) e *N*(*A*) da Aplicação MPEG

As quantidades de alternâncias $M(A)$ de acesso às metades da linha da aplicação MPEG para os *caches* de 8 e 12K *bytes* são detalhadas pelas Tabelas V.13 e V.14, respectivamente, ao passo que o número $N(A)$ de blocos distintos acessados é apresentada na Tabela V.15.

Tabela V.13 *M*(*A*) das matrizes da aplicação MPEG para *cache* de 8K *bytes*.

A	*M*(*A*)								
	b=64 *bytes*			*b*=32 *bytes*			*b*=16 *bytes*		
	n=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2	*n*=8	*n*=4	*n*=2
auxframe	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
auxframe_0	6,82	6,82	6,13	16,67	16,67	16,67	6,26	6,26	6,26
auxframe_1	3,18	1,93	1,93	3,13	3,13	3,13	6,26	6,26	6,26
auxframe_2	1,93	1,89	1,93	4,01	4,01	3,13	6,67	6,67	6,67
backward_reference_frame	0,00	0,00	0,00	0,00	0,00	0,00	40,66	40,66	34,07
backward_reference_frame_0	1,82	2,61	6,26	7,15	8,34	10,72	33,34	35,72	35,72
backward_reference_frame_1	2,78	0,00	0,00	4,17	3,58	3,58	15,39	14,29	14,29
backward_reference_frame_2	1,79	2,09	0,00	8,01	4,01	4,01	15,39	15,39	14,29
base	6,26	6,26	6,26	12,51	12,51	12,51	25,01	25,01	25,01
BMBtab0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
BMBtab1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
c	0,00	0,00	0,00	25,00	25,00	24,98	49,99	49,99	49,98
CBPtab0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	5,56	5,56
Clip	6,26	6,26	6,26	12,51	12,51	12,51	25,01	25,01	25,01
current_frame	0,00	0,00	0,00	52,95	52,95	51,97	0,00	0,00	0,00
dc_dct_pred	0,00	0,00	0,00	0,00	0,00	0,00	31,26	31,26	31,26
DCchromtab0	0,00	0,00	0,00	16,67	16,67	16,67	25,01	25,01	25,01
DCchromtab1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
DClumtab0	18,76	18,76	6,26	25,01	25,01	0,00	0,00	0,00	0,00
DCTtab0	14,59	14,59	14,59	19,05	9,53	11,77	10,01	0,00	7,70
DCTtab0a	16,99	16,49	18,61	18,19	18,19	18,19	16,67	16,67	18,76
DCTtab1	0,00	0,00	0,00	8,34	8,34	8,34	12,51	12,51	12,51
DCTtab1a	25,01	25,01	25,01	40,01	40,01	40,01	0,00	0,00	0,00
DCTtab2	14,29	14,29	14,29	10,01	30,01	30,01	0,00	25,01	0,00
DCTtab3	0,00	8,34	0,00	40,01	40,01	40,01	0,00	0,00	0,00
DCTtab6	0,00	0,00	0,00	11,12	0,00	11,12	0,00	0,00	0,00
DCTtabfirst	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
DCTtabnext	20,41	19,39	19,39	29,63	24,08	24,08	25,01	22,73	22,73
default_intra_quantizer_matrix	8,34	8,34	8,34	0,00	0,00	0,00	25,01	25,01	25,01
dmvector	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
enhan	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
f_code	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
forward_reference_frame	45,91	46,73	46,73	0,00	0,00	0,00	0,00	0,00	0,00
forward_reference_frame_0	6,82	6,53	6,53	15,39	16,67	16,67	25,01	33,34	33,34
forward_reference_frame_1	4,09	2,05	3,78	10,35	4,55	6,90	21,43	20,01	21,43
forward_reference_frame_2	4,77	0,00	2,18	12,51	5,01	5,56	25,01	12,51	12,51
frame_rate_Table	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Motion_vertical_field_select	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Non_Linear_quantizer_scale	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
obfr	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
outname	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
PMBtab0	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
PMBtab1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
PMV	0,00	0,00	0,00	28,12	27,44	27,67	28,84	28,23	28,23
scan	16,48	14,12	14,12	1,57	1,57	0,00	50,01	48,79	47,23
stwclass_table	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Table_6_20	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
tmp	1,39	1,39	1,39	5,55	5,55	5,54	11,10	11,10	11,10
tmpname	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00

Tabela V.14 *M*(*A*) das matrizes da aplicação MPEG para *cache* de 12K *bytes*.

A	*M*(*A*)					
	b=64 *bytes*		*b*=32 *bytes*		*b*=16 *bytes*	
	n=6	*n*=3	*n*=6	*n*=3	*n*=6	*n*=3
auxframe	0,00	0,00	0,00	0,00	0,00	0,00
auxframe_0	6,82	6,82	16,67	16,67	6,26	6,26
auxframe_1	3,71	3,18	3,13	3,13	6,26	6,26
auxframe_2	1,93	1,93	4,17	4,17	6,67	7,15
backward_reference_frame	0,00	0,00	0,00	0,00	40,66	40,66
backward_reference_frame_0	5,36	5,36	10,72	10,72	35,72	35,72
backward_reference_frame_1	4,17	4,17	6,26	6,26	16,67	16,67
backward_reference_frame_2	1,79	2,09	8,01	8,01	15,39	15,39
base	6,26	6,26	12,51	12,51	25,01	25,01
BMBtab0	0,00	0,00	0,00	0,00	0,00	0,00
BMBtab1	0,00	0,00	0,00	0,00	0,00	0,00
c	0,00	0,00	25,00	25,00	49,99	49,99
CBPtab0	0,00	0,00	0,00	0,00	5,56	5,56
Clip	6,26	6,26	12,51	12,51	25,01	25,01
current_frame	0,00	0,00	52,95	52,95	0,00	0,00
dc_dct_pred	0,00	0,00	0,00	0,00	31,26	31,26
DCchromtab0	0,00	0,00	16,67	16,67	25,01	25,01
DCchromtab1	0,00	0,00	0,00	0,00	0,00	0,00
DClumtab0	18,76	18,76	25,01	25,01	0,00	0,00
DCTtab0	14,59	14,59	14,29	14,29	10,01	10,01
DCTtab0a	18,61	18,61	18,19	18,19	16,67	16,67
DCTtab1	0,00	0,00	8,34	8,34	12,51	12,51
DCTtab1a	25,01	25,01	40,01	40,01	0,00	0,00
DCTtab2	14,29	14,29	30,01	30,01	25,01	25,01
DCTtab3	8,34	8,34	40,01	40,01	0,00	0,00
DCTtab6	0,00	0,00	11,12	11,12	0,00	0,00
DCTtabfirst	0,00	0,00	0,00	0,00	0,00	0,00
DCTtabnext	19,39	19,39	29,63	25,93	25,01	22,73
default_intra_quantizer_matrix	8,34	8,34	0,00	0,00	25,01	25,01
dmvector	0,00	0,00	0,00	0,00	0,00	0,00
enhan	0,00	0,00	0,00	0,00	0,00	0,00
f_code	0,00	0,00	0,00	0,00	0,00	0,00
forward_reference_frame	46,73	46,73	0,00	0,00	0,00	0,00
forward_reference_frame_0	9,44	9,26	18,76	18,76	35,72	35,72
forward_reference_frame_1	6,13	6,13	11,54	13,05	25,01	25,01
forward_reference_frame_2	8,58	7,15	13,05	16,67	25,01	30,01
frame_rate_Table	0,00	0,00	0,00	0,00	0,00	0,00
motion_vertical_field_select	0,00	0,00	0,00	0,00	0,00	0,00
non_linear_quantizer_scale	0,00	0,00	16,67	16,67	0,00	0,00
obfr	0,00	0,00	0,00	0,00	0,00	0,00
outname	0,00	0,00	0,00	0,00	0,00	0,00
PMBtab0	0,00	0,00	0,00	0,00	0,00	0,00
PMBtab1	0,00	0,00	0,00	0,00	0,00	0,00
PMV	0,00	0,00	28,12	28,12	28,84	28,84
scan	16,48	14,12	1,57	1,57	50,01	50,01
stwclass_table	0,00	0,00	0,00	0,00	0,00	0,00
Table_6_20	0,00	0,00	0,00	0,00	0,00	0,00
tmp	1,39	1,39	5,55	5,55	11,10	11,10
tmpname	0,00	0,00	0,00	0,00	0,00	0,00

Tabela V.15 $N(A)$ das matrizes da aplicação MPEG.

A	$N(A)$		
	b=64 *bytes*	b=32 *bytes*	b=16 *bytes*
auxframe	2	2	2
auxframe_0	33	65	129
auxframe_1	9	17	33
auxframe_2	8	16	32
backward_reference_frame	1	1	1
backward_reference_frame_0	32	64	128
backward_reference_frame_1	9	17	32
backward_reference_frame_2	9	16	32
base	53	104	206
BMBtab0	1	1	1
BMBtab1	1	1	1
c	16	16	32
CBPtab0	1	1	1
Clip	17	33	64
current_frame	1	1	2
dc_dct_pred	1	1	1
DCchromtab0	1	1	1
DCchromtab1	1	1	1
DClumtab0	1	2	3
DCTtab0	3	6	12
DCTtab0a	12	23	53
DCTtab1	1	1	2
DCTtab1a	1	2	3
DCTtab2	1	2	3
DCTtab3	1	3	3
DCTtab6	1	1	2
DCTtabfirst	1	1	1
DCTtabnext	1	2	3
default_intra_quantizer_matrix	2	3	4
dmvector	1	1	1
enhan	1	1	1
f_code	2	2	2
forward_reference_frame	1	2	2
forward_reference_frame_0	33	65	129
forward_reference_frame_1	9	17	32
forward_reference_frame_2	9	17	33
frame_rate_Table	1	1	1
motion_vertical_field_select	2	2	2
non_Linear_quantizer_scale	1	1	2
obfr	1	1	1
outname	2	2	2
PMBtab0	1	1	1
PMBtab1	1	1	1
PMV	1	1	2
scan	3	5	8
stwclass_table	1	1	1
Table_6_20	1	1	1
tmp	9	16	32
tmpname	1	1	1

Apêndice VI

Avaliação de *Caches* pelo Tempo Médio de Acesso à Memória

Neste apêndice apresentamos os resultados das fases de *Particionamento de Caches*, *Simulação de Cache II*, para *caches* de matrizes particionados, e *Avaliação de Caches* do nosso algoritmo, descritos nas seções 4.2.4, 4.2.5 e 4.2.6 do Capítulo 4, respectivamente.

VI.1 Resultado dos *Caches* Particionados

Para cada par ($C_1|C_2$) simulado, mostramos os resultados da *Simulação de cache II* nas Tabelas VI.1 a VI.4.

Notamos que os valores de *MLIM* diminuem conforme o tamanho de linha aumenta, pois, quanto maior a linha, mais esparso precisa ser o padrão de acesso para que ocorram muitas alternâncias de acesso às metades da linha de *cache*.

Percebemos também que não há relação entre as razões f_1/f_2 e c_1/c_2. Ou seja, o fato de uma partição de *cache* dispor de um espaço de memória maior não implica numa fração maior de dados sendo acessados. Este comportamento pode ser explicado pelo critério para a divisão do espaço de memória do nosso algoritmo, que, ao invés de se basear na *quantidade de acessos* aos blocos mapeados para cada partição de *cache*, baseia-se na *quantidade de blocos distintos* mapeados para cada partição de *cache*.

O número de pares de *caches* ($C_1|C_2$) produzidos tende a aumentar com o número de matrizes acessadas e o grau de associatividade, devido ao maior número de combinações possíveis de configuração de *caches* de matrizes particionados e *MLIM*.

Notamos que não é possível prever o valor de *MLIM* que resulta no menor tempo médio de acesso à memória dos *caches* de matrizes particionados, pois não existe uma tendência de variação do tempo médio de acesso à memória com o valor de *MLIM*.

No entanto, na Tabela 5.6 do Capítulo 5, notamos que as frações de acesso f_1 e f_2 e a configuração de tamanho de linha dos *caches* de matrizes particionados tende a se manter aproximadamente igual nos *caches* de 8 e 12K *bytes*, exceto pela aplicação MPEG, o que pode constituir um indício que *MLIM* é uma métrica promissora.

Tabela VI.1 Simulações dos *caches* de matrizes particionados da aplicação Convolução para *caches* de 8 e 12K *bytes*.

$(b_1, n_1, m_1)\|(b_2, n_2, m_2)$	*MLIM* (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
c = 8K *bytes*												
(64, 7, 16) | (32, 1, 32)	0,00	68	36,89	7	438	1,93	32	0,07	1	1	0,78	6,13
(64, 4, 16) | (32, 4, 32)	6,26	44	22,78	4	219	1,54	56	53,66	4	220	1,38	6,92
(64, 3, 32) | (32, 1, 64)	0,00	68	36,89	6	438	1,68	32	0,07	2	1	0,78	5,95
(64, 2, 32) | (32, 2, 64)	6,26	44	22,78	4	219	1,42	56	53,66	4	220	1,28	6,80
(64, 1, 64) | (32, 1, 128)	0,00	68	36,89	4	438	0,73	32	0,07	4	1	0,76	5,28
(64, 1, 64) | (32, 1, 128)	6,26	44	22,78	4	219	0,73	56	54,68	4	220	0,76	6,26
(32, 4, 32) | (16, 4, 64)	12,5	76	26,44	4	439	1,38	24	250,14	4	437	1,27	9,27
(32, 2, 64) | (16, 2, 128)	12,5	76	26,44	4	439	1,28	24	250,14	4	437	1,32	9,20
(32, 1, 128) | (16, 1, 256)	12,5	76	32,85	4	439	0,76	24	250,14	4	437	0,84	9,25
(16, 4, 64) | (8, 4, 128)	25,0	76	52,82	4	877	1,27	24	500,00	4	874	1,29	14,09
(16, 2, 128) | (8, 2, 256)	25,0	76	52,82	4	877	1,32	24	500,00	4	874	1,29	14,13
(16, 1, 256) | (8, 1, 512)	25,0	76	56,97	4	877	0,84	24	500,00	4	874	0,88	13,88
c = 12K *bytes*												
(64, 5, 32) | (32, 1, 64)	0,00	68	36,89	10	438	1,97	32	0,07	2	1	0,78	6,16
(64, 3, 32) | (32, 3, 64)	6,26	44	22,78	6	219	1,57	56	53,66	6	220	1,37	6,92
(64, 2, 64) | (32, 1, 128)	0,00	68	36,89	8	438	1,50	32	0,07	4	1	0,76	5,82
(64, 1, 64) | (32, 2, 128)	6,26	44	22,78	4	219	0,73	56	53,66	8	220	1,31	6,51
(32, 3, 64) | (16, 3, 128)	12,5	76	26,44	6	439	1,37	24	250,14	6	437	1,34	9,28
(32, 2, 128) | (16, 1, 256)	12,5	76	26,44	8	439	1,31	24	250,14	4	437	0,84	9,08
(16, 3, 128) | (8, 3, 256)	25,0	76	52,82	6	877	1,34	24	500,00	6	874	1,31	14,15
(16, 2, 256) | (8, 1, 512)	25,0	76	52,82	8	877	1,39	24	500,00	4	874	0,88	14,04

Tabela VI.2 Simulações dos *caches* de matrizes particionados da aplicação FFT para *caches* de 8 e 12K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	$MLIM$ (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
c = 8K *bytes*												
(64, 7, 16) | (32, 1, 32)	5,10	10	14,38	4	227	1,93	90	57,03	4	39	0,78	4,99
(64, 3, 16) | (32, 5, 32)	6,42	91	60,37	5	99	1,21	9	15,00	3	167	1,43	4,10
(64, 3, 32) | (32, 1, 64)	5,10	77	18,07	4	227	1,57	23	4,81	4	39	0,78	4,35
(64, 1, 32) | (32, 3, 64)	6,37	91	75,18	6	99	0,71	9	11,95	2	167	1,37	3,91
(64, 1, 64) | (32, 1, 128)	4,87	77	68,11	4	227	0,73	23	1,30	4	39	0,76	11,33
(64, 1, 64) | (32, 1, 128)	6,32	91	33,05	4	99	0,73	9	61,60	4	167	0,76	8,06
(32, 5, 32) | (16, 3, 64)	7,51	77	15,28	4	337	1,43	23	146,23	4	194	1,27	4,25
(32, 4, 32) | (16, 4, 64)	11,46	91	31,95	5	259	1,38	9	63,69	3	272	1,27	5,67
(32, 3, 64) | (16, 1, 128)	7,64	90	12,22	5	337	1,37	10	166,59	3	194	0,78	3,97
(32, 2, 64) | (16, 2, 128)	8,34	76	26,09	4	259	1,28	24	59,50	4	272	1,32	4,96
(32, 1, 128) | (16, 1, 256)	8,53	91	61,91	6	337	0,76	9	77,36	2	194	0,84	8,21
(32, 1, 128) | (16, 1, 256)	11,46	90	67,23	6	259	0,76	10	54,83	2	272	0,84	8,19
(16, 5, 64) | (8, 3, 128)	10,01	76	27,08	4	673	1,31	24	195,93	4	388	1,30	4,85
(16, 5, 64) | (8, 3, 128)	29,92	91	26,73	4	668	1,31	9	177,94	4	393	1,30	4,81
(16, 4, 64) | (8, 4, 128)	38,12	90	60,40	4	512	1,27	10	80,13	4	549	1,29	6,93
(16, 3, 128) | (8, 1, 256)	11,54	76	22,46	4	673	1,34	24	219,42	4	388	0,78	4,59
(16, 3, 128) | (8, 1, 256)	29,49	10	22,12	4	668	1,34	90	201,97	4	393	0,78	4,57
(16, 2, 128) | (8, 2, 256)	38,12	91	49,62	5	512	1,32	9	77,11	3	549	1,29	6,16
(16, 1, 256) | (8, 1, 512)	11,54	77	77,07	4	673	0,84	23	109,21	4	388	0,88	7,95
(16, 1, 256) | (8, 1, 512)	29,49	91	76,78	6	668	0,84	9	103,20	2	393	0,88	7,90
c = 12K *bytes*												
(64, 5, 32) | (32, 1, 64)	5,50	86	3,12	10	227	1,97	14	4,81	2	39	0,78	2,37
(64, 2, 32) | (32, 4, 64)	6,67	10	36,42	4	99	1,42	90	1,15	8	167	1,38	2,14
(64, 2, 64) | (32, 1, 128)	5,89	86	9,13	8	227	1,50	14	1,30	4	39	0,76	2,84
(64, 1, 64) | (32, 2, 128)	6,59	10	33,05	4	99	0,73	90	2,91	8	167	1,31	2,14
(32, 4, 64) | (16, 2, 128)	10,42	91	1,51	8	337	1,38	9	94,53	4	194	1,32	2,28
(32, 3, 64) | (16, 3, 128)	11,46	77	10,26	6	259	1,37	23	24,83	6	272	1,34	2,84
(32, 2, 128) | (16, 1, 256)	10,42	91	3,25	8	337	1,31	9	77,36	4	194	0,84	2,23
(32, 1, 128) | (16, 2, 256)	13,55	77	67,23	4	259	0,76	23	19,21	8	272	1,39	7,58
(16, 4, 128) | (8, 2, 256)	13,47	91	2,61	8	673	1,41	9	135,83	4	388	1,29	2,51
(16, 4, 128) | (8, 2, 256)	23,83	90	2,15	8	668	1,41	10	124,25	4	393	1,29	2,48
(16, 3, 128) | (8, 3, 256)	38,61	76	19,62	6	512	1,34	24	31,19	6	549	1,31	3,26
(16, 2, 256) | (8, 1, 512)	13,47	91	5,31	8	673	1,39	9	109,21	4	388	0,88	2,50
(16, 2, 256) | (8, 1, 512)	29,49	90	4,83	8	668	1,39	10	103,20	4	393	0,88	2,49
(16, 1, 256) | (8, 2, 512)	38,20	76	82,13	4	512	0,84	24	25,75	8	549	1,39	7,13

Tabela VI.3 Simulações dos *caches* de matrizes particionados da aplicação G3fax para *caches* de 8 e 12K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	M L I M (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
c = 8K *bytes*												
(64, 6, 16) | (32, 2, 32)	0,00	79	1,05	6	255	1,87	21	64,55	2	101	9,08	3,40
(64, 1, 16) | (32, 7, 32)	6,26	76	53,58	1	28	10,40	24	12,39	7	341	3,26	8,71
(64, 3, 32) | (32, 1, 64)	0,00	79	1,05	6	255	1,76	21	107,84	2	101	13,80	4,31
(64, 1, 64) | (32, 1, 128)	0,00	79	9,43	4	255	2,43	21	67,91	4	101	8,97	3,82
(32, 7, 32) | (16, 1, 64)	0,00	79	2,08	7	255	2,00	21	212,65	1	101	20,04	5,83
(32, 1, 32) | (16, 7, 64)	12,51	76	102,68	1	28	13,18	24	18,39	7	341	3,07	10,79
(32, 3, 64) | (16, 1, 128)	0,00	79	2,08	6	255	1,62	21	37,10	2	101	4,15	2,16
(32, 1, 128) | (16, 1, 256)	0,00	79	9,27	4	255	1,88	21	7,76	4	101	1,54	1,81
(16, 7, 64) | (8, 1, 128)	0,00	79	4,15	7	255	1,77	21	210,00	1	101	16,69	4,93
(16, 1, 64) | (8, 7, 128)	25,01	76	196,45	1	28	18,57	24	30,98	7	341	3,71	15,06
(16, 3, 128) | (8, 1, 256)	0,00	79	4,15	6	255	1,72	21	36,60	2	101	3,55	2,11
(16, 1, 256) | (8, 1, 512)	0,00	79	11,28	4	255	1,86	21	7,48	4	101	1,45	1,78
c = 12K *bytes*												
(64, 4, 32) | (32, 2, 64)	0,00	79	1,05	8	255	2,00	21	7,16	4	101	2,15	2,03
(64, 2, 64) | (32, 1, 128)	0,00	79	1,15	8	255	1,71	21	67,91	4	101	8,97	3,25
(32, 5, 64) | (16, 1, 128)	0,00	79	2,08	10	255	1,76	21	37,10	2	101	4,15	2,27
(32, 1, 64) | (16, 5, 128)	12,51	76	0,22	2	28	0,80	24	17,56	10	341	3,04	1,33
(32, 2, 128) | (16, 1, 256)	0,00	79	2,11	8	255	1,56	21	7,76	4	101	1,54	1,56
(16, 5, 128) | (8, 1, 256)	0,00	79	4,15	10	255	1,81	21	36,60	2	101	3,55	2,18
(16, 1, 128) | (8, 5, 256)	25,01	76	0,43	2	28	0,82	24	30,15	10	341	3,71	1,50
(16, 2, 256) | (8, 1, 512)	0,00	79	4,16	8	255	1,77	21	7,48	4	101	1,45	1,71

Tabela VI.4 Simulações dos *caches* de matrizes particionados da aplicação JPEG para *caches* de 8 e 12 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	M L I M (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
c = 8K *bytes*												
(64, 7, 16) | (32, 1, 32)	0,00	93	1,38	7	271	1,93	7	0,06	1	2	0,78	2,08
(64, 2, 16) | (32, 6, 32)	6,26	91	1,24	2	76	1,15	9	21,01	6	197	1,47	1,62
(64, 3, 32) | (32, 1, 64)	0,00	93	1,38	6	272	1,57	7	0,06	2	2	0,78	1,74
(64, 3, 32) | (32, 1, 64)	6,67	92	1,16	6	206	1,57	8	3,84	2	68	0,78	1,74
(64, 1, 32) | (32, 3, 64)	7,82	91	1,24	2	77	0,71	9	21,01	6	197	1,37	1,22
(64, 1, 64) | (32, 1, 128)	0,00	93	3,19	4	272	0,73	7	0,06	4	2	0,76	1,27
(64, 1, 64) | (32, 1, 128)	6,67	92	3,05	4	206	0,73	8	3,84	4	68	0,76	1,28
(64, 1, 64) | (32, 1, 128)	7,82	91	1,09	4	77	0,73	9	21,01	4	197	0,76	1,15
(32, 7, 32) | (16, 1, 64)	0,00	61	4,15	7	534	1,75	39	0,03	1	6	0,74	1,66
(32, 4, 32) | (16, 4, 64)	9,38	59	1,40	4	277	1,38	41	8,14	4	263	1,27	1,74
(32, 2, 32) | (16, 6, 64)	12,51	59	1,00	2	145	1,26	41	9,48	6	395	1,35	1,72
(32, 3, 64) | (16, 1, 128)	0,00	61	4,15	6	534	1,37	39	0,03	2	6	0,78	1,45
(32, 1, 64) | (16, 3, 128)	12,51	59	1,00	2	145	0,78	41	9,48	6	395	1,34	1,43
(32, 1, 128) | (16, 1, 256)	0,00	61	5,10	4	534	0,76	39	0,03	4	6	0,84	1,17
(32, 1, 128) | (16, 1, 256)	12,51	59	0,54	4	145	0,76	41	9,60	4	395	0,84	1,19
(16, 7, 64) | (8, 1, 128)	0,00	93	5,44	7	1064	1,39	7	0,06	1	2	0,78	1,80
(16, 7, 64) | (8, 1, 128)	11,11	61	8,25	7	1060	1,39	39	0,06	1	6	0,78	1,61
(16, 6, 64) | (8, 2, 128)	25,01	61	7,11	6	796	1,35	39	2,95	2	270	1,31	1,82
(16, 4, 64) | (8, 4, 128)	33,34	60	5,59	4	539	1,27	40	7,97	4	527	1,29	1,83
(16, 3, 128) | (8, 1, 256)	0,00	93	5,44	6	1064	1,34	7	0,06	2	2	0,78	1,76
(16, 3, 128) | (8, 1, 256)	11,11	61	8,25	6	1060	1,34	39	0,06	2	6	0,78	1,58
(16, 3, 128) | (8, 1, 256)	25,01	61	6,91	6	796	1,34	39	2,95	2	270	0,78	1,59
(16, 2, 128) | (8, 2, 256)	33,34	60	5,59	4	539	1,32	40	7,28	4	527	1,29	1,83
(16, 1, 256) | (8, 1, 512)	0,00	93	5,91	4	1064	0,84	7	0,06	4	2	0,88	1,34
(16, 1, 256) | (8, 1, 512)	11,11	61	8,89	4	1060	0,84	39	0,06	4	6	0,88	1,35
(16, 1, 256) | (8, 1, 512)	25,01	61	7,97	4	796	0,84	39	2,95	4	270	0,88	1,38
(16, 1, 256) | (8, 1, 512)	37,51	59	1,04	4	284	0,84	41	19,19	4	782	0,88	1,51
c = 12 K *bytes*												
(64, 5, 32) | (32, 1, 64)	0,00	93	0,78	10	272	1,97	7	0,06	2	2	0,78	2,02
(64, 5, 32) | (32, 1, 64)	6,67	92	0,62	10	206	1,97	8	3,84	2	68	0,78	2,02
(64, 2, 32) | (32, 4, 64)	7,82	91	0,19	4	197	1,42	9	9,31	8	77	1,38	1,55
(64, 2, 64) | (32, 1, 128)	0,00	93	0,96	8	272	1,50	7	0,06	4	2	0,76	1,61
(64, 2, 64) | (32, 1, 128)	6,67	92	0,79	8	206	1,50	8	3,84	4	68	0,76	1,61
(64, 1, 64) | (32, 2, 128)	7,82	91	1,09	4	197	0,73	9	9,21	8	77	1,31	1,07
(32, 5, 64) | (16, 1, 128)	0,00	61	2,26	10	534	1,51	39	0,03	2	6	0,78	1,39
(32, 4, 64) | (16, 2, 128)	12,51	61	2,62	8	402	1,38	39	1,48	4	138	1,32	1,60
(32, 2, 64) | (16, 4, 128)	15,63	59	0,54	4	145	1,28	41	4,27	8	395	1,41	1,53
(32, 2, 128) | (16, 1, 256)	0,00	61	2,55	8	534	1,31	39	0,03	4	6	0,84	1,31
(32, 2, 128) | (16, 1, 256)	12,51	61	2,06	8	402	1,31	39	1,48	4	138	0,84	1,33
(32, 1, 128) | (16, 2, 256)	15,63	59	0,54	4	145	0,76	41	4,18	8	395	1,39	1,22
(16, 5, 128) | (8, 1, 256)	0,00	93	2,94	10	1064	1,43	7	0,06	2	2	0,78	1,63
(16, 5, 128) | (8, 1, 256)	11,11	61	4,42	10	1060	1,43	39	0,06	2	6	0,78	1,42
(16, 4, 128) | (8, 2, 256)	25,01	61	5,07	8	796	1,41	39	2,95	4	270	1,29	1,74
(16, 3, 128) | (8, 3, 256)	41,67	60	5,59	6	539	1,34	40	5,89	6	527	1,31	1,81
(16, 2, 256) | (8, 1, 512)	0,00	93	3,31	8	1064	1,39	7	0,06	4	2	0,88	1,64
(16, 2, 256) | (8, 1, 512)	11,11	61	4,98	8	1060	1,39	39	0,06	4	6	0,88	1,47
(16, 2, 256) | (8, 1, 512)	25,01	61	4,00	8	796	1,39	39	2,95	4	270	0,88	1,50
(16, 2, 256) | (8, 1, 512)	41,67	60	2,27	8	539	1,39	40	5,89	4	527	0,88	1,49

Tabela VI.5 Simulações dos *caches* de matrizes particionados de tamanho de linha de 64 e 32 *bytes* da aplicação de MPEG para *cache* de 8 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	$MLIM$ (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(64, 7, 16) | (32, 1, 32)	0,00	47	4,05	7	255	2,67	53	1,82	1	46	1,00	1,78
(64, 7, 16) | (32, 1, 32)	1,39	20	8,51	7	246	3,48	80	96,01	1	55	12,37	10,55
(64, 6, 16) | (32, 2, 32)	1,79	20	8,98	6	237	3,31	80	1,22	2	64	1,40	1,79
(64, 5, 16) | (32, 3, 32)	1,82	20	9,59	5	205	3,25	80	1,07	3	96	1,48	1,83
(64, 5, 16) | (32, 3, 32)	1,93	20	8,42	5	197	3,04	80	1,20	3	104	1,49	1,80
(64, 5, 16) | (32, 3, 32)	2,78	20	6,88	5	188	2,76	80	1,55	3	113	1,54	1,78
(64, 5, 16) | (32, 3, 32)	3,18	20	6,33	5	179	2,66	80	1,77	3	122	1,56	1,78
(64, 5, 16) | (32, 3, 32)	4,09	20	3,85	5	170	2,21	80	2,09	3	131	1,60	1,72
(64, 4, 16) | (32, 4, 32)	4,77	20	4,17	4	161	2,30	80	1,52	4	140	1,57	1,71
(64, 2, 16) | (32, 6, 32)	6,26	1	103,30	2	91	19,87	99	1,26	6	210	1,62	1,76
(64, 1, 16) | (32, 7, 32)	6,82	1	101,39	1	25	19,04	99	1,90	7	276	1,98	2,01
(64, 1, 16) | (32, 7, 32)	8,34	1	101,08	1	23	18,98	99	1,91	7	278	1,98	2,01
(64, 1, 16) | (32, 7, 32)	14,29	1	99,49	1	22	18,69	99	1,92	7	279	1,98	2,01
(64, 1, 16) | (32, 7, 32)	14,59	1	66,85	1	19	12,79	99	1,97	7	282	1,99	2,00
(64, 3, 32) | (32, 1, 64)	0,00	47	4,32	6	238	2,35	53	1,70	2	64	0,98	1,62
(64, 3, 32) | (32, 1, 64)	1,39	20	9,09	6	229	3,22	80	48,70	2	73	6,66	5,96
(64, 3, 32) | (32, 1, 64)	1,89	20	7,56	6	221	2,94	80	48,75	2	81	6,67	5,91
(64, 3, 32) | (32, 1, 64)	1,93	20	6,56	6	212	2,76	80	49,19	2	90	6,72	5,92
(64, 3, 32) | (32, 1, 64)	2,05	20	4,63	6	203	2,41	80	49,63	2	99	6,77	5,90
(64, 3, 32) | (32, 1, 64)	2,09	20	3,21	6	194	2,15	80	50,05	2	108	6,82	5,89
(64, 2, 32) | (32, 2, 64)	2,61	20	4,80	4	162	2,29	80	2,17	4	140	1,54	1,69
(64, 1, 32) | (32, 3, 64)	6,26	1	100,08	2	92	18,80	99	2,01	6	210	1,61	1,74
(64, 1, 32) | (32, 3, 64)	6,53	1	50,79	2	59	9,89	99	2,69	6	243	1,70	1,74
(64, 1, 64) | (32, 1, 128)	0,00	47	8,44	4	237	2,25	53	1,15	4	65	0,90	1,53
(64, 1, 64) | (32, 1, 128)	1,39	20	15,79	4	228	3,58	80	1,13	4	74	0,90	1,45
(64, 1, 64) | (32, 1, 128)	1,93	20	13,57	4	211	3,18	80	1,35	4	91	0,93	1,38
(64, 1, 64) | (32, 1, 128)	2,18	20	11,66	4	202	2,84	80	1,77	4	100	0,98	1,35
(64, 1, 64) | (32, 1, 128)	3,78	20	4,05	4	193	2,56	80	1,82	4	109	1,02	1,33
(64, 1, 64) | (32, 1, 128)	6,13	19	8,51	4	160	1,93	81	96,01	4	142	1,23	1,37

Tabela VI.6 Simulações dos *caches* de matrizes particionados de tamanho de linha de 32 e 16 *bytes* da aplicação de MPEG para *cache* de 8 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	M L I M (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(32, 5, 32) | (16, 3, 64)	0,00	67	2,23	5	513	1,70	33	2,83	3	30	1,53	1,64
(32, 5, 32) | (16, 3, 64)	1,57	67	2,21	5	508	1,70	33	2,84	3	35	1,53	1,64
(32, 5, 32) | (16, 3, 64)	3,13	67	2,20	5	491	1,70	33	2,86	3	52	1,54	1,64
(32, 5, 32) | (16, 3, 64)	4,01	67	1,54	5	475	1,62	33	3,25	3	68	1,57	1,60
(32, 5, 32) | (16, 3, 64)	4,17	67	1,52	5	458	1,61	33	3,27	3	85	1,57	1,60
(32, 3, 32) | (16, 5, 64)	5,55	48	1,36	3	442	1,51	52	3,09	5	101	1,59	1,55
(32, 2, 32) | (16, 6, 64)	7,15	48	1,43	2	378	1,43	52	3,96	6	165	1,71	1,58
(32, 1, 32) | (16, 7, 64)	8,01	47	1,18	1	361	0,92	53	4,09	7	182	1,76	1,36
(32, 6, 32) | (16, 2, 64)	8,34	94	3,36	6	360	1,78	6	0,98	2	183	0,87	1,72
(32, 6, 32) | (16, 2, 64)	10,01	94	3,26	6	358	1,76	6	1,19	2	185	0,89	1,71
(32, 6, 32) | (16, 2, 64)	10,35	94	2,99	6	341	1,73	6	2,16	2	202	0,98	1,69
(32, 6, 32) | (16, 2, 64)	11,12	94	2,57	6	340	1,68	6	5,74	2	203	1,30	1,66
(32, 6, 32) | (16, 2, 64)	12,51	94	2,08	6	186	1,62	6	11,56	2	357	1,83	1,63
(32, 6, 32) | (16, 2, 64)	15,39	94	1,59	6	121	1,56	6	14,52	2	422	2,10	1,60
(32, 6, 32) | (16, 2, 64)	16,67	93	1,26	6	55	1,52	7	20,64	2	488	2,66	1,60
(32, 3, 64) | (16, 1, 128)	0,00	67	1,57	6	511	1,56	33	5,04	2	31	1,24	1,45
(32, 2, 64) | (16, 2, 128)	1,57	67	2,50	6	506	1,58	33	3,73	2	36	1,66	1,61
(32, 2, 64) | (16, 2, 128)	3,13	67	2,36	6	489	1,56	33	3,99	2	53	1,68	1,60
(32, 1, 64) | (16, 3, 128)	3,58	48	3,16	6	472	1,16	52	3,47	2	70	1,66	1,42
(32, 2, 64) | (16, 2, 128)	4,01	94	12,60	6	440	2,29	6	1,13	2	102	0,94	2,20
(32, 3, 64) | (16, 1, 128)	4,55	94	12,13	6	423	2,23	6	2,95	2	119	1,11	2,16
(32, 3, 64) | (16, 1, 128)	5,01	94	11,76	6	406	2,18	6	4,35	2	136	1,23	2,12
(32, 3, 64) | (16, 1, 128)	5,55	94	11,43	6	390	2,14	6	6,42	2	152	1,42	2,10
(32, 3, 64) | (16, 1, 128)	8,34	67	14,63	4	325	2,53	33	1,74	4	217	1,00	2,03
(32, 2, 64) | (16, 2, 128)	9,53	67	13,96	4	319	2,45	33	2,14	4	223	1,03	1,98
(32, 1, 64) | (16, 3, 128)	12,51	67	13,52	2	182	2,40	33	2,58	6	360	1,07	1,96
(32, 1, 64) | (16, 3, 128)	0,00	67	13,51	4	505	2,40	33	2,59	4	37	1,07	1,96
(32, 1, 64) | (16, 3, 128)	3,13	67	12,92	4	472	2,32	33	3,15	4	70	1,12	1,93
(32, 1, 128) | (16, 1, 256)	3,58	67	12,91	4	455	2,32	33	4,13	4	87	1,21	1,95
(32, 1, 128) | (16, 1, 256)	4,01	67	12,90	4	439	2,32	33	4,44	4	103	1,24	1,96
(32, 1, 128) | (16, 1, 256)	5,54	48	1,77	4	423	0,98	52	5,78	4	119	1,36	1,18
(32, 1, 128) | (16, 1, 256)	5,56	67	2,23	4	406	1,70	33	2,83	4	136	1,53	1,64
(32, 1, 128) | (16, 1, 256)	6,90	67	2,21	4	389	1,70	33	2,84	4	153	1,53	1,64
(32, 1, 128) | (16, 1, 256)	8,34	67	2,20	4	388	1,70	33	2,86	4	154	1,54	1,64
(32, 1, 128) | (16, 1, 256)	10,72	67	1,54	4	324	1,62	33	3,25	4	218	1,57	1,60
(32, 1, 128) | (16, 1, 256)	11,12	67	1,52	4	323	1,61	33	3,27	4	219	1,57	1,60
(32, 1, 128) | (16, 1, 256)	11,77	48	1,36	4	317	1,51	52	3,09	4	225	1,59	1,55

Tabela VI.7 Simulações dos *caches* de matrizes particionados de tamanho de linha de 16 e 8 *bytes* da aplicação de MPEG para *cache* de 8 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1, n_1, m_1)\|(b_2, n_2, m_2)$	$MLIM$ (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(16, 7, 64) | (8, 1, 128)	0,00	94	2,49	7	992	1,62	6	1,54	1	43	0,89	1,57
(16, 6, 64) | (8, 2, 128)	6,26	94	2,49	6	830	1,58	6	10,94	2	205	2,14	1,61
(16, 6, 64) | (8, 2, 128)	6,67	94	2,42	6	798	1,57	6	13,18	2	237	2,32	1,62
(16, 6, 64) | (8, 2, 128)	10,01	93	2,35	6	786	1,56	7	13,61	2	249	2,35	1,61
(16, 6, 64) | (8, 2, 128)	11,10	67	3,00	6	754	1,62	33	4,06	2	281	1,62	1,62
(16, 6, 64) | (8, 2, 128)	12,51	67	2,99	6	752	1,62	33	4,07	2	283	1,62	1,62
(16, 5, 64) | (8, 3, 128)	15,39	67	3,17	5	688	1,60	33	3,83	3	347	1,59	1,60
(16, 5, 64) | (8, 3, 128)	16,67	67	2,92	5	645	1,58	33	4,19	3	390	1,62	1,59
(16, 5, 64) | (8, 3, 128)	21,43	67	2,45	5	613	1,53	33	4,77	3	422	1,66	1,58
(16, 1, 64) | (8, 7, 128)	25,01	47	2,87	1	172	1,00	53	6,26	7	863	1,82	1,43
(16, 1, 64) | (8, 7, 128)	28,84	47	2,85	1	170	1,00	53	6,29	7	865	1,82	1,43
(16, 1, 64) | (8, 7, 128)	31,26	47	2,69	1	169	0,99	53	6,30	7	866	1,82	1,43
(16, 3, 128) | (8, 1, 256)	0,00	94	3,58	6	984	1,67	6	1,66	2	50	0,90	1,62
(16, 3, 128) | (8, 1, 256)	5,56	94	3,57	6	983	1,67	6	1,72	2	51	0,91	1,62
(16, 3, 128) | (8, 1, 256)	6,26	94	2,93	6	821	1,61	6	10,99	2	213	1,61	1,61
(16, 3, 128) | (8, 1, 256)	6,67	93	2,83	6	789	1,60	7	13,35	2	245	1,79	1,61
(16, 3, 128) | (8, 1, 256)	11,10	67	3,76	6	757	1,68	33	4,95	2	277	1,15	1,51
(16, 3, 128) | (8, 1, 256)	12,51	67	3,09	6	722	1,62	33	5,85	2	312	1,22	1,49
(16, 3, 128) | (8, 1, 256)	14,29	67	2,77	6	690	1,59	33	6,91	2	344	1,30	1,50
(16, 3, 128) | (8, 1, 256)	15,39	67	2,47	6	658	1,57	33	7,66	2	376	1,36	1,50
(16, 2, 128) | (8, 2, 256)	16,67	67	3,92	4	615	1,67	33	4,95	4	419	1,67	1,67
(16, 2, 128) | (8, 2, 256)	20,01	67	3,29	4	583	1,62	33	5,75	4	451	1,73	1,65
(16, 2, 128) | (8, 2, 256)	22,73	67	3,19	4	580	1,61	33	5,79	4	454	1,73	1,65
(16, 1, 128) | (8, 3, 256)	25,01	48	4,06	2	301	1,15	52	5,34	6	733	1,71	1,45
(16, 1, 128) | (8, 3, 256)	28,23	48	3,98	2	299	1,14	52	5,45	6	735	1,72	1,45
(16, 1, 128) | (8, 3, 256)	31,26	48	3,94	2	298	1,14	52	5,47	6	736	1,72	1,45
(16, 1, 256) | (8, 1, 512)	0,00	94	13,22	4	1013	2,04	6	1,19	4	41	0,97	1,97
(16, 1, 256) | (8, 1, 512)	5,56	94	13,20	4	1012	2,04	6	1,24	4	42	0,98	1,97
(16, 1, 256) | (8, 1, 512)	6,26	94	11,85	4	850	1,91	6	10,25	4	204	1,66	1,90
(16, 1, 256) | (8, 1, 512)	6,67	94	11,77	4	818	1,91	6	11,71	4	236	1,77	1,90
(16, 1, 256) | (8, 1, 512)	7,70	93	11,67	4	786	1,90	7	12,14	4	268	1,80	1,89
(16, 1, 256) | (8, 1, 512)	11,10	67	15,40	4	754	2,24	33	4,39	4	300	1,22	1,90
(16, 1, 256) | (8, 1, 512)	12,51	67	14,73	4	719	2,18	33	4,85	4	335	1,25	1,87
(16, 1, 256) | (8, 1, 512)	14,29	67	14,05	4	655	2,11	33	6,37	4	399	1,37	1,87
(16, 1, 256) | (8, 1, 512)	18,76	67	13,67	4	612	2,08	33	6,73	4	442	1,39	1,85
(16, 1, 256) | (8, 1, 512)	21,43	67	13,34	4	580	2,05	33	7,63	4	474	1,46	1,85
(16, 1, 256) | (8, 1, 512)	22,73	67	13,25	4	577	2,04	33	7,73	4	477	1,47	1,85
(16, 1, 256) | (8, 1, 512)	25,01	48	2,23	4	301	1,04	52	8,80	4	753	1,55	1,31
(16, 1, 256) | (8, 1, 512)	28,23	48	2,19	4	299	1,04	52	8,94	4	755	1,56	1,31
(16, 1, 256) | (8, 1, 512)	31,26	48	2,18	4	298	1,04	52	9,01	4	756	1,57	1,31

Tabela VI.8 Simulações dos *caches* de matrizes particionados de tamanho de linha de 64 e 32 *bytes* da aplicação de MPEG para *cache* de 12 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	M L I M (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(64, 5, 32) | (32, 1, 64)	0,00	47	3,28	10	256	2,57	53	0,38	2	46	0,82	1,64
(64, 5, 32) | (32, 1, 64)	1,39	20	7,17	10	247	3,27	80	47,83	2	55	6,55	5,88
(64, 5, 32) | (32, 1, 64)	1,79	20	5,11	10	238	2,90	80	47,99	2	64	6,57	5,83
(64, 5, 32) | (32, 1, 64)	1,93	20	4,67	10	230	2,82	80	48,04	2	72	6,58	5,82
(64, 4, 32) | (32, 2, 64)	3,71	20	5,83	8	221	2,87	80	0,42	4	81	1,33	1,64
(64, 4, 32) | (32, 2, 64)	4,17	20	4,40	8	212	2,61	80	0,81	4	90	1,38	1,62
(64, 4, 32) | (32, 2, 64)	5,36	20	3,34	8	180	2,42	80	1,42	4	122	1,45	1,64
(64, 4, 32) | (32, 4, 64)	6,13	20	3,09	6	171	2,13	80	0,86	6	131	1,47	1,60
(64, 2, 32) | (32, 4, 64)	6,26	8	51,59	4	101	10,78	99	0,96	8	201	1,50	2,38
(64, 1, 32) | (32, 5, 64)	6,82	1	162,20	2	68	30,03	99	0,87	10	234	1,61	1,76
(64, 1, 32) | (32, 5, 64)	8,34	1	161,13	2	65	29,83	99	0,88	10	237	1,61	1,76
(64, 1, 32) | (32, 5, 64)	8,58	1	114,52	2	56	21,41	99	1,11	10	246	1,64	1,73
(64, 2, 64) | (32, 1, 128)	0,00	47	4,79	8	256	2,37	53	0,32	4	47	0,80	1,54
(64, 2, 64) | (32, 1, 128)	1,39	20	10,18	8	247	3,35	80	0,50	4	56	0,83	1,34
(64, 2, 64) | (32, 1, 128)	1,93	20	9,51	8	239	3,23	80	0,55	4	64	0,83	1,32
(64, 2, 64) | (32, 1, 128)	2,09	20	7,36	8	230	2,84	80	0,73	4	73	0,85	1,26
(64, 2, 64) | (32, 1, 128)	3,18	20	6,10	8	221	2,61	80	0,82	4	82	0,86	1,22
(64, 2, 64) | (32, 1, 128)	4,17	20	5,00	8	212	2,41	80	1,34	4	91	0,93	1,22
(64, 2, 64) | (32, 1, 128)	5,36	20	4,46	8	180	2,31	80	2,11	4	123	1,02	1,27
(64, 2, 64) | (32, 1, 128)	6,13	20	4,40	8	171	2,30	80	2,48	4	132	1,06	1,31
(64, 1, 64) | (32, 2, 128)	6,26	1	85,47	4	101	16,18	99	1,24	8	202	1,46	1,58
(64, 1, 64) | (32, 2, 128)	6,82	1	92,07	4	68	17,37	99	1,62	8	235	1,50	1,59
(64, 1, 64) | (32, 2, 128)	7,15	1	36,03	4	59	7,24	99	1,77	8	244	1,52	1,55
(64, 1, 64) | (32, 2, 128)	8,34	1	35,32	4	56	7,11	99	1,79	8	247	1,53	1,55

Tabela VI.9 Simulações dos *caches* de matrizes particionados de tamanho de linha de 32 e 16 *bytes* da aplicação de MPEG para *cache* de 12 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	$MLIM$ (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(32, 5, 64) | (16, 1, 128)	0,00	94	1,60	10	514	1,70	6	0,79	2	29	0,86	1,65
(32, 5, 64) | (16, 1, 128)	1,57	94	1,55	10	509	1,70	6	1,00	2	34	0,87	1,65
(32, 5, 64) | (16, 1, 128)	3,13	94	1,40	10	492	1,68	6	1,97	2	51	0,96	1,63
(32, 5, 64) | (16, 1, 128)	4,17	94	1,33	10	476	1,67	6	5,84	2	67	1,31	1,65
(32, 5, 64) | (16, 1, 128)	5,55	67	1,81	10	460	1,73	33	1,65	2	83	0,93	1,47
(32, 5, 64) | (16, 1, 128)	6,26	67	1,57	10	443	1,70	33	2,47	2	100	1,01	1,47
(32, 5, 64) | (16, 1, 128)	8,01	67	1,24	10	427	1,66	33	2,92	2	116	1,05	1,46
(32, 5, 64) | (16, 1, 128)	8,34	67	1,24	10	426	1,66	33	2,93	2	117	1,05	1,46
(32, 4, 64) | (16, 2, 128)	10,72	67	1,27	8	362	1,54	33	2,40	4	181	1,54	1,54
(32, 4, 64) | (16, 2, 128)	11,12	67	1,25	8	361	1,54	33	2,41	4	182	1,54	1,54
(32, 4, 64) | (16, 2, 128)	11,54	67	0,89	8	344	1,49	33	2,81	4	199	1,57	1,52
(32, 2, 64) | (16, 4, 128)	12,51	48	1,35	4	207	1,44	52	2,35	8	336	1,63	1,54
(32, 2, 64) | (16, 4, 128)	13,05	48	0,89	4	190	1,39	52	2,67	8	353	1,66	1,53
(32, 2, 64) | (16, 4, 128)	14,29	48	0,84	4	184	1,38	52	2,81	8	359	1,67	1,53
(32, 1, 64) | (16, 5, 128)	16,67	47	2,10	2	117	1,03	53	3,00	10	426	1,71	1,39
(32, 1, 64) | (16, 5, 128)	18,19	47	1,62	2	94	0,97	53	3,13	10	449	1,72	1,36
(32, 2, 128) | (16, 1, 256)	0,00	94	2,61	8	514	1,62	6	0,79	4	29	0,91	1,58
(32, 2, 128) | (16, 1, 256)	1,57	94	2,49	8	509	1,61	6	1,00	4	34	0,93	1,57
(32, 2, 128) | (16, 1, 256)	3,13	94	2,26	8	492	1,58	6	1,92	4	51	1,01	1,55
(32, 2, 128) | (16, 1, 256)	4,17	94	2,17	8	476	1,57	6	2,82	4	67	1,09	1,54
(32, 2, 128) | (16, 1, 256)	5,55	67	2,83	8	460	1,65	33	0,80	4	83	0,91	1,41
(32, 2, 128) | (16, 1, 256)	6,26	67	2,46	8	443	1,61	33	1,25	4	100	0,95	1,39
(32, 2, 128) | (16, 1, 256)	8,01	67	2,16	8	427	1,57	33	1,66	4	116	0,99	1,38
(32, 2, 128) | (16, 1, 256)	8,34	67	2,16	8	426	1,57	33	1,67	4	117	0,99	1,38
(32, 2, 128) | (16, 1, 256)	10,72	67	1,88	8	362	1,54	33	3,12	4	181	1,12	1,40
(32, 2, 128) | (16, 1, 256)	11,12	67	1,88	8	361	1,54	33	3,15	4	182	1,12	1,40
(32, 1, 128) | (16, 2, 256)	12,51	48	3,03	4	224	1,13	52	2,22	8	319	1,60	1,37
(32, 1, 128) | (16, 2, 256)	13,05	48	2,54	4	207	1,07	52	2,46	8	336	1,62	1,36
(32, 1, 128) | (16, 2, 256)	14,29	48	2,52	4	201	1,07	52	2,56	8	342	1,63	1,36
(32, 1, 128) | (16, 2, 256)	16,67	47	1,19	4	117	0,91	53	3,59	8	426	1,72	1,34
(32, 1, 128) | (16, 2, 256)	18,19	47	1,03	4	94	0,89	53	3,72	8	449	1,74	1,33
(32, 4, 64) | (16, 2, 128)	11,54	67	0,89	8	344	1,49	33	2,81	4	199	1,57	1,52

Tabela VI.10 Simulações dos *caches* de matrizes particionados de tamanho de linha de 16 e 8 *bytes* da aplicação de MPEG para *cache* de 12 K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	M L I M (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(16, 5, 128) | (8, 1, 256)	0,00	94	2,21	10	1005	1,63	6	1,11	2	38	0,86	1,59
(16, 5, 128) | (8, 1, 256)	5,56	94	2,21	10	1004	1,63	6	1,17	2	39	0,87	1,59
(16, 5, 128) | (8, 1, 256)	6,26	94	1,87	10	842	1,60	6	10,48	2	201	1,57	1,60
(16, 5, 128) | (8, 1, 256)	6,67	94	1,81	10	810	1,60	6	12,85	2	233	1,75	1,61
(16, 5, 128) | (8, 1, 256)	10,01	93	1,74	10	798	1,59	7	13,35	2	245	1,79	1,60
(16, 4, 128) | (8, 2, 256)	11,10	67	2,69	8	766	1,66	33	2,58	4	277	1,49	1,60
(16, 4, 128) | (8, 2, 256)	12,51	67	2,69	8	764	1,66	33	2,60	4	279	1,49	1,60
(16, 4, 128) | (8, 2, 256)	15,39	67	2,38	8	732	1,63	33	3,09	4	311	1,53	1,60
(16, 4, 128) | (8, 2, 256)	16,67	67	2,03	8	647	1,60	33	4,12	4	396	1,61	1,60
(16, 2, 128) | (8, 4, 256)	25,01	48	1,35	4	301	1,44	52	5,07	8	742	1,77	1,61
(16, 2, 128) | (8, 4, 256)	28,84	48	1,36	4	299	1,44	52	5,09	8	744	1,77	1,61
(16, 2, 128) | (8, 4, 256)	31,26	48	1,35	4	298	1,44	52	5,10	8	745	1,77	1,61
(16, 2, 256) | (8, 1, 512)	0,00	94	3,14	8	996	1,68	6	1,06	4	38	0,96	1,64
(16, 2, 256) | (8, 1, 512)	5,56	94	3,13	8	995	1,68	6	1,11	4	39	0,97	1,64
(16, 2, 256) | (8, 1, 512)	6,26	94	2,53	8	833	1,63	6	10,13	4	201	1,65	1,63
(16, 2, 256) | (8, 1, 512)	7,15	94	2,47	8	801	1,62	6	11,59	4	233	1,76	1,63
(16, 2, 256) | (8, 1, 512)	10,01	93	2,41	8	789	1,62	7	12,02	4	245	1,80	1,63
(16, 2, 256) | (8, 1, 512)	11,10	67	3,23	8	757	1,69	33	4,37	4	277	1,22	1,53
(16, 2, 256) | (8, 1, 512)	12,51	67	3,22	8	755	1,69	33	4,38	4	279	1,22	1,53
(16, 2, 256) | (8, 1, 512)	15,39	67	2,95	8	723	1,66	33	4,93	4	311	1,26	1,53
(16, 2, 256) | (8, 1, 512)	16,67	67	2,56	8	648	1,63	33	6,26	4	386	1,36	1,54
(16, 2, 256) | (8, 1, 512)	22,73	67	2,52	8	645	1,63	33	6,34	4	389	1,36	1,54
(16, 1, 256) | (8, 2, 512)	25,01	48	2,98	4	334	1,11	52	4,84	8	700	1,76	1,45
(16, 1, 256) | (8, 2, 512)	28,84	48	2,94	4	332	1,11	52	4,86	8	702	1,76	1,45
(16, 1, 256) | (8, 2, 512)	30,01	48	2,19	4	299	1,04	52	5,15	8	735	1,78	1,43
(16, 1, 256) | (8, 2, 512)	31,26	48	2,18	4	298	1,04	52	5,16	8	736	1,78	1,43

Apêndice VII

Aplicação Iterativa do Algoritmo

Neste apêndice apresentamos os resultados do particionamento iterativo dos *caches* descrito na seção 4.2.7 do Capítulo 4.

VII.1 *M*(*A*) e *N*(*A*) das Aplicações no Particionamento Iterativo de *Caches*

As Tabelas VII.1 a VII.6 apresentam a fração *M*(*A*) de alternâncias de acesso às metades de linha e o número *N*(*A*) de blocos distintos acessados de cada matriz *A* nas aplicações durante a segunda iteração do nosso algoritmo.

Tabela VII.1 *M*(*A*) e *N*(*A*) das matrizes da aplicação Convolução para *cache* de 8K *bytes*, *b* = 64 *bytes*, *n* = 2 e *m* = 64 que realizam segunda iteração do *cache* C_1 produzido pela simulação de *cache* de 12K *bytes*, *b* = 64 *bytes*, *n* = 3, *m* = 64 e *MLIM* = 0 %.

A	*M*(*A*)	*N*(*A*)
z	6,26	219
y	11,37	219

Tabela VII.2 *M*(*A*) e *N*(*A*) das matrizes da aplicação FFT para *cache* de 6K *bytes*, *b* = 32 *bytes*, *n* = 3 e *m* = 64 que realizam a segunda iteração do *cache* C_2 produzido pela simulação de *cache* de 8K *bytes*, *b* = 64 *bytes*, *n* = 4, *m* = 32 e *MLIM* = 6,37 %.

A	*M*(*A*)	*N*(*A*)
t	10,42	78
a	14,65	256

Tabela VII.3 $M(A)$ e $N(A)$ das matrizes da aplicação FFT para *cache* de 8K *bytes*, $b = 32$ *bytes*, $n = 2$ e $m = 128$ que realizam a segunda iteração do *cache* C_2 produzido pela simulação de *cache* de 12K *bytes*, $b = 64$ *bytes*, $n = 3$, $m = 64$ e $MLIM = 6{,}59$ %.

A	$M(A)$	$N(A)$
t	13,55	78
a	15,22	256

Tabela VII.4 $M(A)$ e $N(A)$ das matrizes da aplicação G3fax para *cache* de 10K *bytes*, $b = 16$ *bytes*, $n = 5$ e $m = 128$ que realizam a segunda iteração do *cache* C_2 produzido pela simulação de *cache* de 12K *bytes*, $b = 32$ *bytes*, $n = 6$, $m = 64$ e $MLIM = 12{,}51$ %.

A	$M(A)$	$N(A)$
fax	25,01	955
g3white	0,00	23
g3black	0	93

Tabela VII.5 $M(A)$ e $N(A)$ das matrizes da aplicação JPEG para *cache* de 8K *bytes*, $b = 32$ *bytes*, $n = 2$ e $m = 128$ que realizam a segunda iteração do *cache* C_2 produzido pela simulação de *cache* de 12K *bytes*, $b = 64$ *bytes*, $n = 3$, $m = 64$ e $MLIM = 7{,}82$ %.

A	$M(A)$	$N(A)$
imageFileName	0,00	1
imageFileHandle	0,00	1
buffer	12,51	132
buffer3	15,63	257

Tabela VII.6 $M(A)$ e $N(A)$ das matrizes da aplicação MPEG para *cache* de 8K *bytes*, b = 64 *bytes*, n = 2 e m = 64 que realizam a segunda iteração do *cache* C_1 produzido pela simulação de *cache* de 12K *bytes*, b = 64 *bytes*, n = 3, m = 64 e *MLIM* = 3,18 %.

A	$M(A)$	$N(A)$
auxframe_0	6,82	33
backward_reference_frame_0	5,46	32
backward_reference_frame_1	4,69	9
base	6,26	53
Clip	6,26	17
DClumtab0	18,76	1
DCTtab0	14,59	3
DCTtab0a	18,61	12
DCTtab1a	25,01	1
DCTtab2	14,29	1
DCTtab3	8,34	1
DCTtabnext	20,41	1
default_intra_quantizer_matrix	8,34	2
forward_reference_frame	46,73	1
forward_reference_frame_0	9,44	33
forward_reference_frame_1	6,13	9
forward_reference_frame_2	2,39	9
scan	14,12	3

VII.2 Resultados do Particionamento Iterativo de *Caches*

As Tabelas VII.7 a VII.11 apresentam os resultados das simulações iterativas.

Tabela VII.7 Simulações dos *caches* de matrizes particionados da segunda iteração a partir de C_1 gerado na primeira iteração da aplicação Convolução no *cache* de 12K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1, n_1, m_1) \mid (b_2, n_2, m_2)$	*MLIM* (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1 \mid C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(64, 1, 64) | (32, 1, 128)	6,26	65	22,78	4	219	4,85	35	125,1	4	219	15,87	8,74

Tabela VII.8 Simulações dos *caches* de matrizes particionados da segunda iteração a partir de C_2 gerado na primeira iteração da aplicação FFT. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	MLIM (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
Cache de 8K *bytes*												
(32, 2, 64) | (16, 1, 128)	10,42	84	25,88	4	256	1,28	16	9,35	2	78	0,78	3,98
Cache de 12K *bytes*												
(32, 1, 128) | (16, 1, 256)	13,55	84	67,26	4	256	0,76	16	2,60	4	78	0,84	7,67

Tabela VII.9 Simulações dos *caches* de matrizes particionados da segunda iteração a partir de C_2 gerado na primeira iteração da aplicação G3fax no *cache* de 12K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	MLIM (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(16, 4, 128) | (8, 1, 256)	0	10	125,08	8	955	12,85	90	36,6	2	116	3,55	4,48

Tabela VII.10 Simulações dos *caches* de matrizes particionados da segunda iteração a partir de C_2 gerado na primeira iteração da aplicação JPEG no *cache* de 12K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	MLIM (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(32, 1, 128) | (16, 1, 256)	0,00	22	97,29	4	389	12,51	78	0,06	4	2	0,84	3,36
(32, 1, 128) | (16, 1, 256)	12,51	19	94,12	4	257	12,13	81	7,62	4	134	1,53	3,55

Tabela VII.11 Simulações dos *caches* de matrizes particionados da segunda iteração a partir de C_1 gerado na primeira iteração da aplicação MPEG no *cache* de 12K *bytes*. c_1 e c_2 em K *bytes*.

$(b_1,\ n_1,\ m_1)\|(b_2,\ n_2,\ m_2)$	MLIM (%)	C_1					C_2					Tempo médio de acesso à memória $(C_1\|C_2)$ (ns)
		f_1 (%)	Taxa de Falta (x1000)	c_1	No. blocos acessados das matrizes em C_1	Tempo médio de acesso à memória (ns)	f_2 (%)	Taxa de Falta (x1000)	c_2	No. blocos acessados das matrizes em C_2	Tempo médio de acesso à memória (ns)	
(64, 1, 64) | (32, 1, 128)	2,39	99	13,32	4	212	3,14	1	37,86	4	9	5,34	3,14
(64, 1, 64) | (32, 1, 128)	4,69	99	11,73	4	203	2,85	1	35,64	4	18	5,07	2,87
(64, 1, 64) | (32, 1, 128)	5,46	98	10,15	4	171	2,56	2	72,60	4	50	9,53	2,71
(64, 1, 64) | (32, 1, 128)	6,13	97	9,19	4	162	2,39	3	95,48	4	59	12,29	2,64
(64, 1, 64) | (32, 1, 128)	6,26	4	47,62	4	92	9,34	96	6,33	4	129	1,53	1,82
(64, 1, 64) | (32, 1, 128)	6,82	2	36,03	4	59	7,24	98	8,66	4	162	1,81	1,93
(64, 1, 64) | (32, 1, 128)	8,34	2	35,32	4	56	7,11	98	8,76	4	165	1,82	1,94

Apêndice VIII

Avaliação de *Caches* pelo Consumo de Energia

Neste apêndice apresentamos os resultados da avaliação de *caches* descrita na seção 4.2.6 do Capítulo 4, com a diferença que o critério de seleção dos *caches* de matrizes unificados e particionados é baseado no menor consumo de energia.

VIII.1 Resultados da Avaliação de *Caches* Baseado no *Cache* de Menor Consumo de Energia

Para cada aplicação, a Tabela VIII.1 descreve as organizações de *cache* de matrizes unificado e particionado de menor consumo de energia para *caches* de 8 e 12K *bytes*.

Tabela VIII.1 *Caches* de matrizes unificados e particionados com os menores consumos de energia. Organização de *cache*: (tamanho de linha, associatividade, número de conjuntos).

Aplicação	$(b,\ n,\ m)$	$(b_1,\ n_1,\ m_1)$ | $(b_2,\ n_2,\ m_2)$
c = 8K *bytes*		
Conv	(32,2,128)	(64,1,64) | (32,1,128)
FFT	(32,2,128)	(16,3,128) | (8,1,256)
G3fax	(16,2,256)	(16,3,128) | (8,1,256)
JPEG	(16,2,256)	(16,3,128) | (8,1,256)
MPEG	(16,2,256)	(16,1,256) | (8,1,512)
c = 12K *bytes*		
Conv	(32,3,128)	(64,1,64) | (32,2,128)
FFT	(16,3,256)	(16,2,256) | (8,1,512)
G3fax	(16,3,256)	(16,2,256) | (8,1,512)
JPEG	(16,3,256)	(16,2,256) | (8,1,512)
MPEG	(16,3,256)	(16,2,256) | (8,1,512)

A Figura VIII.1 ilustra a redução no consumo de energia dos *caches* de matrizes particionados sobre os unificados.

A Figura VIII.2 compara os *caches* de matrizes unificados e particionados da Tabela VIII.1, em termos de tempo médio de acesso à memória. Como esperado, a vantagem dos *caches* de matrizes particionados sobre os unificados aumenta com a fração dos acessos paralelos, para todas as aplicações.

A Figura VIII.3 mostra a redução na taxa de falta dos *caches* de matrizes particionados sobre os unificados. Os valores negativos indicam que a taxa de faltas nos *caches* de matrizes particionados é, em muitos casos, maior que a do *cache* de matrizes unificado.

Nossas comparações entre *caches* de matrizes unificados e particionados dividem a diferença de valores encontrados entre *caches* de matrizes unificados e particionados pelo valor referente ao *cache* de matrizes unificado, conforme especificado pela Equação 5.2 do Capítulo 5.

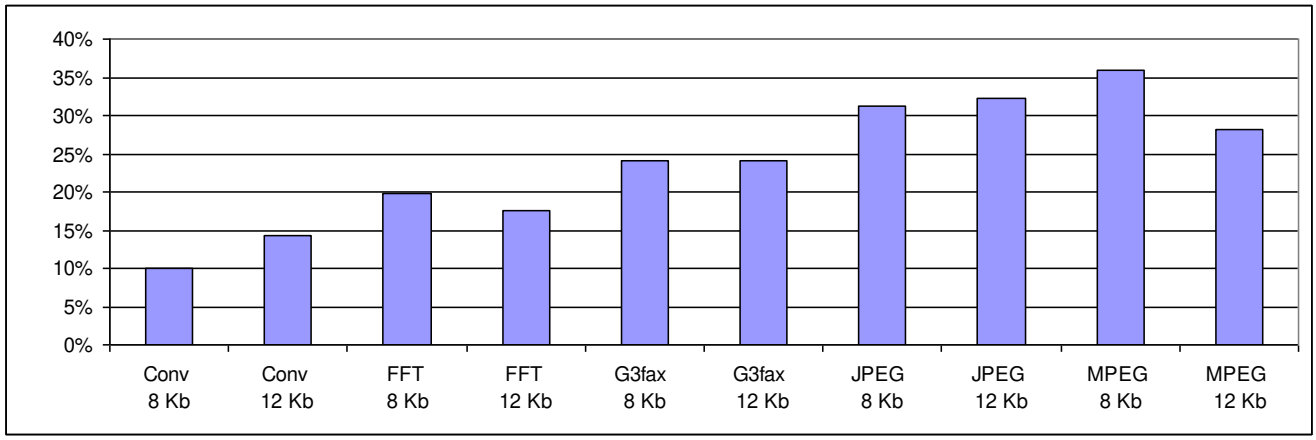


Figura VIII.1 Redução no consumo de energia de *caches* de matrizes particionados de menor consumo de energia baseados em rastro em relação aos *caches* de matrizes unificados de menor consumo de energia.

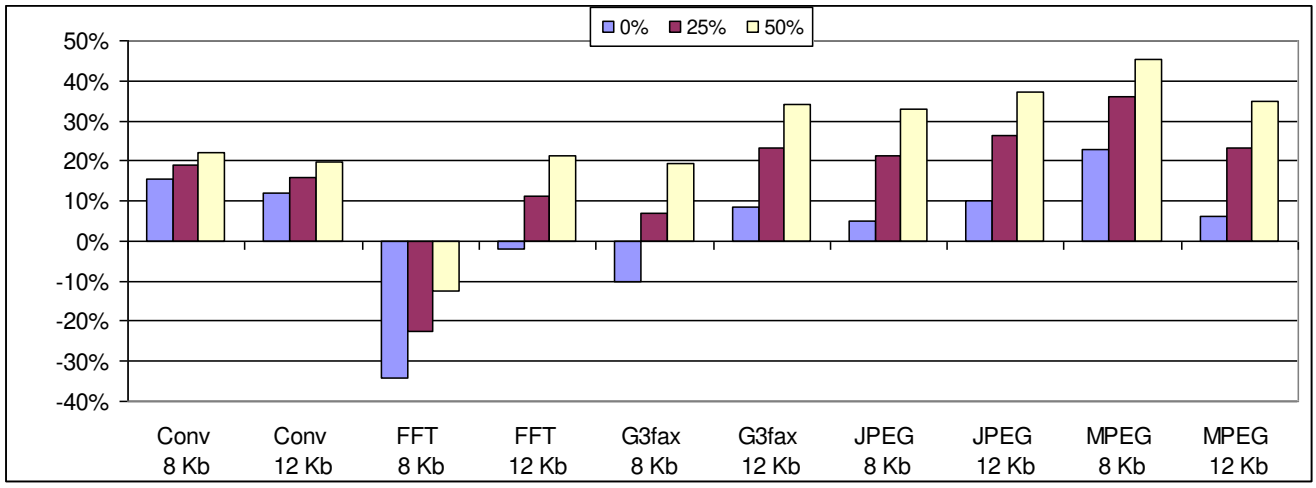


Figura VIII.2 Redução no tempo médio de acesso à memória de *caches* de matrizes particionados de menor consumo de energia baseados em rastro em relação aos *caches* de matrizes unificados de menor consumo de energia para diferentes valores de Φ. Os valores negativos indicam que em alguns casos o tempo médio de acesso à memória dos *caches* de matrizes particionados é maior que a do *cache* de matrizes unificado.

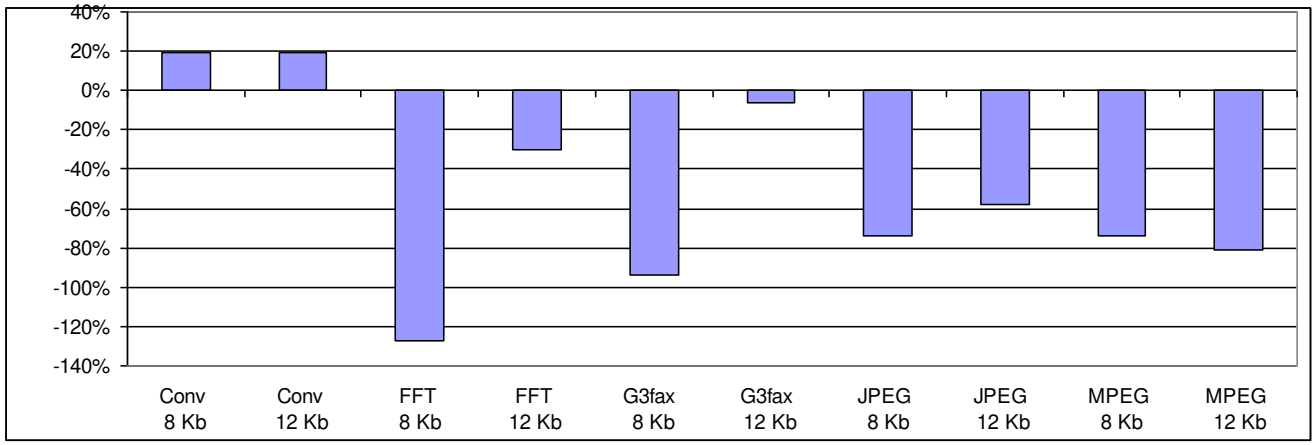


Figura VIII.3 Redução na taxa de faltas de *caches* de matrizes particionados de menor consumo de energia baseados em rastro em relação aos *caches* de matrizes unificados de menor consumo de energia. Os valores negativos indicam que a taxa de faltas nos *caches* de matrizes particionados é, em muitos casos, maior que a do *cache* de matrizes unificado.

Apêndice IX

Avaliação de *Caches* pelo Produto do Consumo de Energia e Tempo Médio de Acesso à Memória

Neste apêndice apresentamos os resultados da avaliação dos *caches* descrita na seção 4.2.6 do Capítulo 4, com a diferença que o critério de seleção dos *caches* de matrizes unificados e particionados é baseado no menor produto do consumo de energia e tempo médio de acesso à memória.

IX.1 Resultados da Avaliação de *Caches* Baseado no *Cache* de Menor Produto de Consumo de Energia e Tempo Médio de Acesso à Memória

Para cada aplicação, a Tabela IX.1 descreve as organizações de *cache* de matrizes unificado e particionado de menor produto consumo de energia e tempo médio de acesso à memória para *caches* de 8 e 12K *bytes*.

A Figura IX.1 ilustra a redução no produto consumo de energia e tempo médio de acesso à memória dos *caches* de matrizes particionados sobre os unificados. As Figuras IX.2, IX.3 e IX.4 comparam os *caches* de matrizes unificados e particionados da Tabela IX.1, em termos de consumo de energia, tempo médio de acesso à memória e taxa de faltas, respectivamente. Como esperado, a vantagem dos *caches* de matrizes particionados sobre os unificados aumenta com a fração dos acessos paralelos, para todas as aplicações.

Tabela IX.1 *Caches* de matrizes unificados e particionados com os menores produtos consumo de energia e tempos médios de acesso à memória. Organização de *cache*: (tamanho de linha, associatividade, número de conjuntos).

Aplicação	$(b,\ n,\ m)$	$(b_1,\ n_1,\ m_1)$ | $(b_2,\ n_2,\ m_2)$
c = 8K *bytes*		
Conv	(32,2,128)	(64,1,64) | (32,1,128)
FFT	(32,2,128)	(16,3,128) | (8,1,256)
G3fax	(16,2,256)	(16,1,256) | (8,1,512)
JPEG	(16,2,256)	(32,1,128) | (16,1,256)
MPEG	(16,2,256)	(16,1,256) | (8,1,512)
c = 12K *bytes*		
Conv	(32,3,128)	(64,1,64) | (32,2,128)
FFT	(16,6,128)	(16,2,256) | (8,1,512)
G3fax	(16,3,256)	(16,1,128) | (8,5,256)
JPEG	(16,3,256)	(16,2,256) | (8,1,512)
MPEG	(16,3,256)	(32,1,128) | (16,2,256)

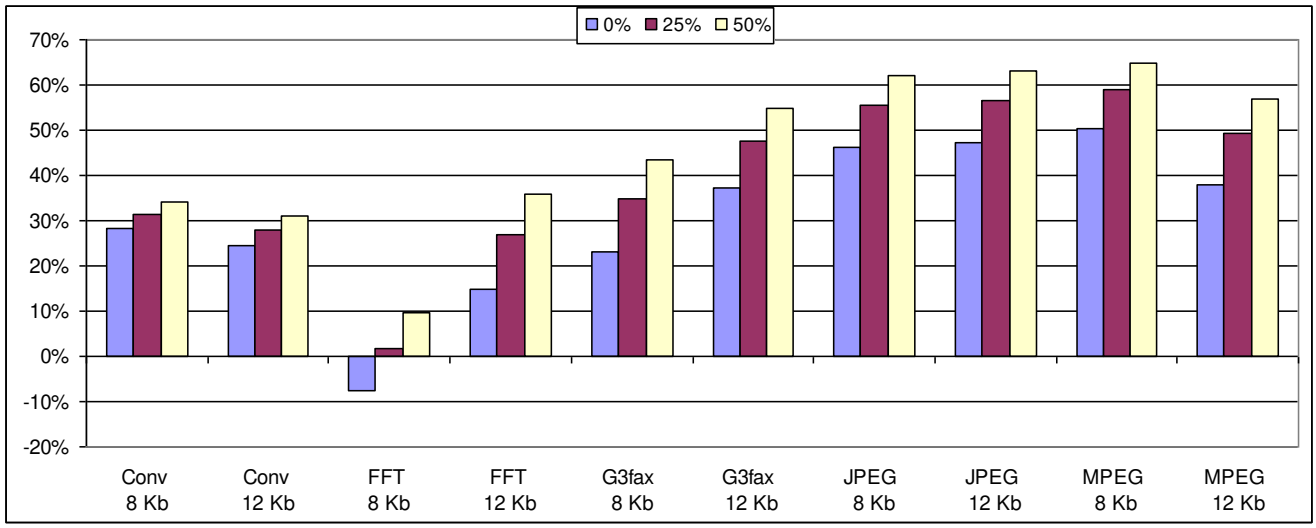


Figura IX.1 Redução no produto do consumo de energia e tempo médio de acesso à memória do *cache* de matrizes particionado de menor produto do consumo de energia e tempo médio de acesso à memória em relação aos *caches* de matrizes unificados de menor produto do consumo de energia e tempo médio de acesso à memória.

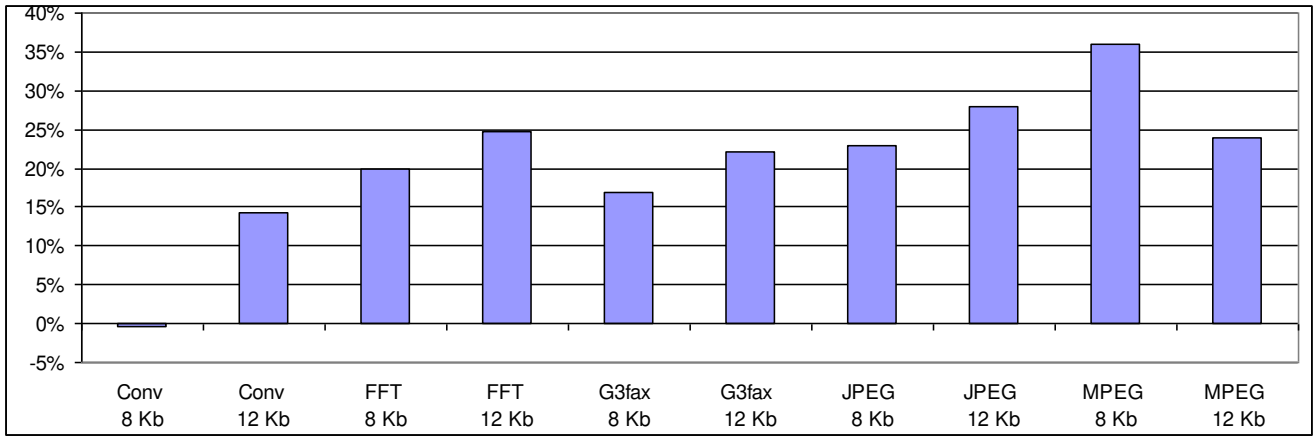


Figura IX.2 Redução no consumo de energia do *cache* de matrizes particionado de menor produto do consumo de energia e tempo médio de acesso à memória em relação aos *caches* de matrizes unificados de menor produto do consumo de energia e tempo médio de acesso à memória.

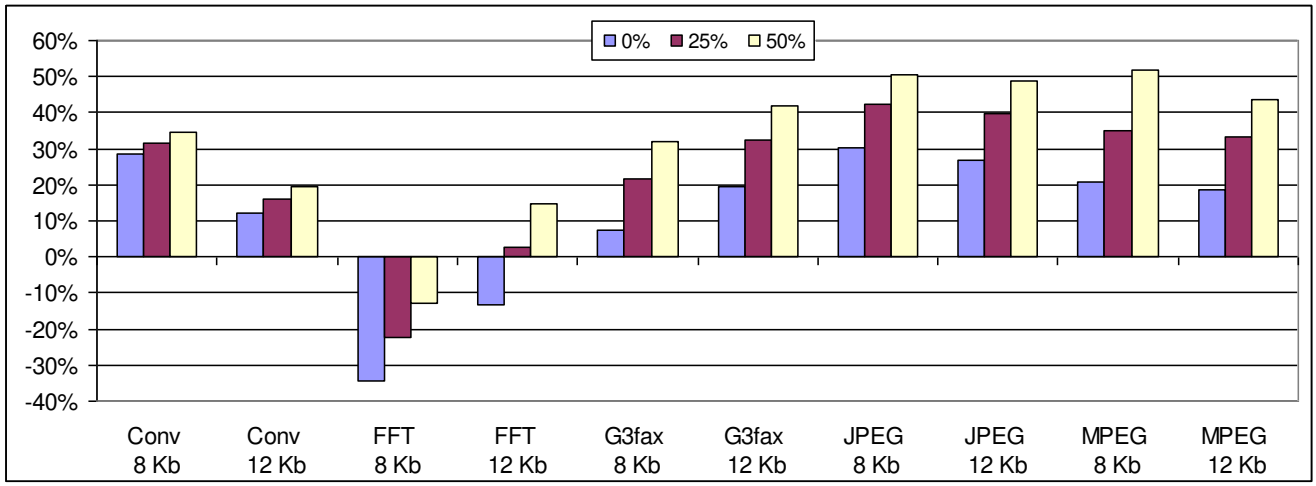


Figura IX.3 Redução no tempo médio de acesso à memória do *cache* de matrizes particionado de menor produto do consumo de energia e tempo médio de acesso à memória em relação aos *caches* de matrizes unificados de menor produto do consumo de energia e tempo médio de acesso à memória para diferentes valores de Φ. Os valores negativos indicam que em alguns casos o tempo médio de acesso à memória dos *caches* de matrizes particionados é maior que a do *cache* de matrizes unificado.

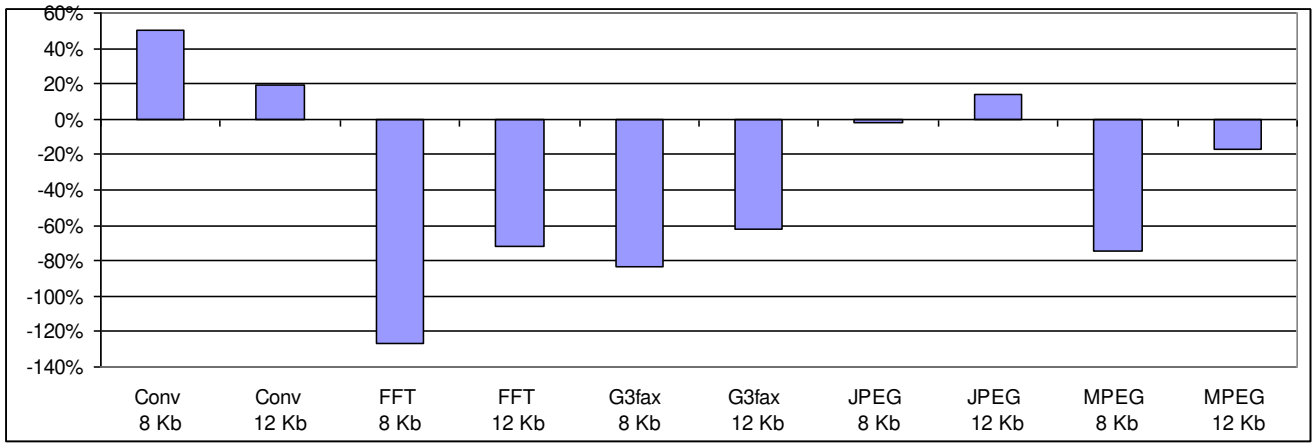


Figura IX.4 Redução na taxa de faltas do *cache* de matrizes particionado de menor produto do consumo de energia e tempo médio de acesso à memória em relação aos *caches* de matrizes unificados de menor produto do consumo de energia e tempo médio de acesso à memória. Os valores negativos indicam que a taxa de faltas nos *caches* de matrizes particionados é, em muitos casos, maior que a do *cache* de matrizes unificado.

Nossas comparações entre *caches* de matrizes unificados e particionados dividem a diferença de valores encontrados entre *caches* de matrizes unificados e particionados pelo valor referente ao *cache* de matrizes unificado, conforme especificado pela Equação 5.2 do Capítulo 5.