GIANCARLO COVOLO HECK

INVESTIGAÇÃO DE TÉCNICAS DE PROJETO DE CACHE DE DADOS PARA SISTEMAS EMBARCADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2008

GIANCARLO COVOLO HECK

INVESTIGAÇÃO DE TÉCNICAS DE PROJETO DE CACHE DE DADOS PARA SISTEMAS EMBARCADOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA

2008

GIANCARLO COVOLO HECK

INVESTIGAÇÃO DE TÉCNICAS DE PROJETO DE CACHE DE DADOS PARA SISTEMAS EMBARCADOS

Dissertação aprovada como requisito parcial à obtenção do grau de Mestre no Programa de Pós-Graduação em Informática da Universidade Federal do Paraná, pela Comisso formada pelos professores:

Orientador: Prof. Dr. Roberto André Hexsel
Departamento de Informática, UFPR

Prof. Dr. Sandro Rigo
Instituto de Computação, UNICAMP

Prof. Dr. Aldri Luiz dos Santos
Departamento de Informática, UFPR

Curitiba, 22 de Agosto de 2008

AGRADECIMENTOS

Agradeço à minha querida e amada esposa Elisane Belniaki pelo incentivo, apoio e companheirismo sempre presentes, pela paciência e, em especial, por abrigar a Fernanda, nossa primeira grande riqueza que ainda este ano plenificará nossa família.

Ao amigo e professor Roberto André Hexsel por sua dedicação e excelente orientação, pela inigualável companhia nos caminhos percorridos e nos que ainda virão e pelos ensinamentos que ultrapassam o espaço acadêmico.

Aos familiares e amigos, pelo apoio e pelos momentos de descontração.

A todos que contribuíram de alguma forma.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A disparidade entre a evolução da velocidade dos microprocessadores e da latência no acesso às memórias torna o projeto da hierarquia de memória um importante fator no desempenho global dos sistemas com microprocessadores.

O projeto de sistemas embarcados é guiado por parâmetros tais como desempenho, consumo de energia, tamanho, confiabilidade, custo e *time to market*. Memórias cache fornecem ganhos de desempenho mas podem ocasionar impactos negativos quanto ao consumo de energia, ao tamanho do circuito e ao custo.

Neste trabalho, o projeto de caches de dados foi investigado com o intuito de obter combinações de parâmetros de projeto que produzam os melhores resultados com as menores capacidades. Os experimentos foram executados no simulador SimpleScalar com os programas de teste do pacote CommBench.

Foi desenvolvida uma nova técnica chamada de Cache de Vítima e Controle de Poluição (*Pollution Control Victim Cache* - PCVC), a qual melhora a Cache de Controle de Poluição (*Pollution Control Cache* - PCC), pois é mais simples e apresenta melhor desempenho. Os resultados indicam que a Cache de Vítimas e o *Stream Buffer* não possuem bom desempenho nos pequenos sistemas simulados. A PCC e a PCVC mostraram resultados promissores.

ABSTRACT

A well designed memory hierarchy is important because of the increasing disparity between the gains in the speed of microprocessors and the small reductions on memory access time.

The design of embedded systems is constrained by several parameters such as performance, power consumption, size, reliability, cost, and time to market. Cache memories provide gains in performance but might cause negative impacts on power consumption, circuit size, and cost.

This work explores the design space of data caches, looking for the combinations of design parameters that produce the best results at the smallest capacities. The simulations were performed with the SimpleScalar suite, running the CommBench benchmarks.

A novel cache algorithm, named Pollution Control Victim Cache (PCVC), was devised and improves on the Pollution Control Cache (PCC). The PCVC is simpler and performs better than the PCC. The simulation results also indicate that Victim Caches and Stream Buffers do not perform well in the small systems that were simulated. Both the PCC and the PCVC have shown promising results.

CAPÍTULO 1

INTRODUÇÃO

Os microprocessadores desenvolvidos após 1980 obtiveram aumentos de desempenho, principalmente quanto à velocidade de processamento, de cerca de 35% ao ano até 1986 e 55% ao ano a partir de 1987 até o ano 2000, quando a taxa retornou aos 35% ao ano. Em contrapartida, neste mesmo período, a redução na latência no acesso à memória foi de 7% ao ano [10]. Esta disparidade entre a evolução das velocidades dos microprocessadores e das memórias torna o projeto da hierarquia de memória um importante fator no desempenho global do sistema.

Conforme enuncia a Lei de Amdahl, o ganho de desempenho obtido com a utilização de um microprocessador mais rápido será limitado pela fração de tempo na qual este é capaz de executar alguma tarefa útil, não havendo ganho de desempenho enquanto ele fica parado à espera de dados provenientes da memória, por exemplo [10]. Desta forma, quando operando com memórias de mesma latência, um processador mais rápido fica parado, proporcionalmente, mais tempo que um processador lento.

Para amenizar esta diferença de desempenho, nas últimas quatro décadas os arquitetos de computadores propuseram inúmeras técnicas e modelos avançados de memória cache, visto que a memória cache é o item da hierarquia que permite diminuir o tempo médio de acesso à memória. Podemos citar dentre os inúmeros exemplos de técnicas e modelos descritos na literatura, (i) a cache não-bloqueante, (ii) a cache de vítima, (iii) os *stream* e *write buffers*, (iv) a cache de controle de poluição, (v) a cache não-temporal e (vi) a cache multi-portas.

Os sistemas embarcados dominam amplamente o mercado de microprocessadores e constituem o setor que cresce mais rapidamente na área de informática. Microprocessadores para sistemas embarcados variam amplamente em poder de processamento e custo, podem processar instruções de 8 bits até 64 bits e custar de centavos até centenas de dólares.

Atualmente, estima-se que mais de 98% do total de microprocessadores de 32 bits fabricados no mundo são utilizados em sistemas embarcados [10] [28] [4].

O projeto de sistemas embarcados é guiado por parâmetros tais como desempenho, consumo de energia, tamanho, confiabilidade, custo e *time to market*. Memórias cache fornecem ganhos de desempenho mas podem ocasionar impactos negativos quanto ao consumo de energia, ao tamanho do circuito e ao custo.

Neste estudo, algumas das técnicas citadas acima foram empregadas — isoladamente ou de maneira combinada — no ambiente dos sistemas embarcados e um grande número de combinações de parâmetros de projeto foram testados com o intuito de obter o melhor desempenho com o menor tamanho de cache. São apresentados resultados de simulação para uma hierarquia de memória formada por uma cache de dados de primeiro nível, com ao menos um circuito especializado, e memória principal (DRAM).

Durante o desenvolvimento do trabalho um novo modelo de memória cache foi concebido e simulado, a Cache de Vítima com Controle de Poluição (*Pollution Control Victim Cache* - PCVC). A PCVC se originou da combinação entre uma cache de vítima e uma de controle de poluição, organizadas de maneira a reduzir o número de circuitos envolvidos, e apresentou os melhores resultados dentre os modelos avaliados. Como resultado deste estudo, o artigo "*The Performance of Pollution Control Victim Cache for Embedded Systems*" foi aceito e apresentado no *21st Symposium on Integrated Circuits and Systems Design* (SBCCI '08)[8].

A avaliação de desempenho foi realizada com a execução de seis programas de teste do pacote CommBench [30] sobre o simulador SimpleScalar [21] [3] [1]. O CommBench é composto por núcleos de programas que realizam computação intensa e refletem aplicações de rede e multimídia, comuns a processadores de rede. O SimpleScalar é um conjunto de simuladores altamente configuráveis que permite a criação de modelos de arquiteturas de microprocessadores e hierarquias de memória, bem como a avaliação de desempenho detalhada destes modelos com a execução de programas reais ou de teste.

O texto está organizado como segue. No Capítulo 2 são expostos os conceitos envolvidos no estudo. No Capítulo 3 são apresentadas as ferramentas e os métodos utilizados durante o desenvolvimento da pesquisa. No Capítulo 4 são descritas algumas das técnicas

de aperfeiçoamento da memória cache propostas na literatura. Para os modelos avaliados no estudo são detalhadas as formas de implementação e são apresentados e discutidos seus desempenhos individuais, bem como são comentados os trabalhos relacionados. A análise de desempenho é apresentada no Capítulo 5 e o Capítulo 6 sintetiza os resultados e discute as possibilidades de trabalhos futuros.

CAPÍTULO 2

CONCEITOS

Neste capítulo são discutidos os principais conceitos envolvidos no estudo, principalmente quanto à memória cache que é o foco do trabalho.

2.1 Sistemas Embarcados

Um sistema embarcado caracteriza-se pelo conjunto *hardware* e *software* que fazem parte de um sistema maior, sistema este que pode operar sem intervenção humana e que geralmente possui propósitos específicos. Pode-se encontrar sistemas embarcados em utensílios domésticos tais como televisores, geladeiras, aparelhos de som, em automóveis, em celulares, em máquinas fotográficas, em equipamentos de automação comercial e industrial, por exemplo [12].

Tipicamente o sistema embarcado utiliza um microprocessador com *software* gravado em memória de leitura (*Read Only Memory* - ROM) e começa a executar sua(s) tarefa(s) assim que é ligado, não parando até que seja desligado [7]. Como o sistema é desenvolvido para executar uma quantidade limitada de tarefas, seus projetistas otimizam os circuitos eletrônicos que o compõem (processadores, memórias, interfaces de comunicação, etc.) com o intuito de reduzir o tamanho e o consumo de energia destes componentes. Além disso, quando são fabricados em larga escala, tais sistemas geralmente apresentam baixo custo de produção.

2.2 Hierarquia de Memória

O conjunto formado pelos vários níveis de memória cache e pela memória principal é chamado de *hierarquia de memória* [10] [11]. Há definições que incluem os registradores do processador e os dispositivos de entrada e saída como parte da hierarquia de memória.

A Figura 2.1 mostra um diagrama de blocos que representa uma hierarquia de memória. Neste estudo, foram consideradas apenas a memória cache de dados de primeiro nível (L1) e a memória principal. Sistemas com um segundo nível de cache (L2) estão fora do escopo deste trabalho.

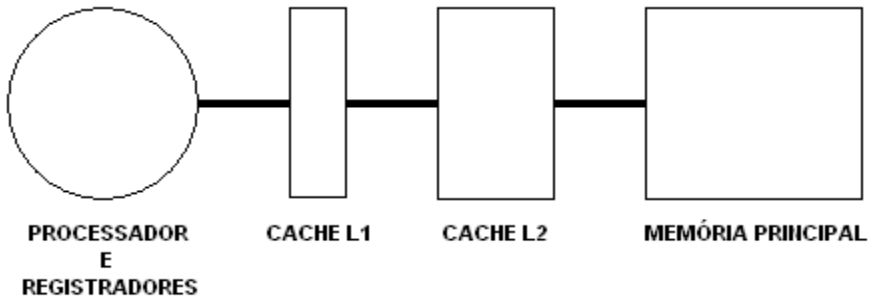

Figura 2.1: Hierarquia de memória.

2.3 Memória Cache

A memória cache, ou simplesmente cache, é uma memória pequena e rápida que armazena um cópia dos dados recentemente acessados pelo processador. Sua função é de reduzir o tempo dispendido pelo processador no acesso aos dados alocados na memória principal [23] [11]. Acessos subseqüentes aos dados previamente copiados para a cache são atendidos com tempo de acesso muito menor que o tempo necessário para acessar a memória principal, sendo a diferença nos tempos de acesso de uma a duas ordens de grandeza.

O ganho de tempo decorre de dois fatores. Caches em geral têm uma capacidade muito menor que a memória principal e são implementadas com circuitos rápidos. O segundo fator é a proximidade do processador. Normalmente utiliza-se um primeiro nível de memória cache (*Cache Level 1* - "L1") dentro do encapsulamento do processador, o que permite a construção de interligações curtas e, consequentemente, a redução do tempo de acesso. Além da L1, costuma-se encontrar em muitos microprocessadores um segundo nível de memória cache (*Cache Level 2* - "L2") e, às vezes, até um terceiro nível (L3) entre a L2 e a memória principal, com capacidade de armazenamento e tempo de acesso um pouco maiores que os da L2.

Localidade Temporal e Espacial

Memórias tiram proveito de dois princípios de localidade, a *localidade temporal*, que postula que um dado acessado recentemente tem grande probabilidade de ser acessado novamente em um futuro próximo, e a *localidade espacial*, que postula que dados em endereços próximos a um dado acessado recentemente têm grande probabilidade de serem acessados em breve.

Os princípios de localidade trazem benefícios ao funcionamento da cache somente se alguns cuidados forem tomados durante a fase de projeto da cache. Os parâmetros de projeto, capacidade, tamanho de bloco e associatividade, se combinados com cuidado, fazem com que os dados recentemente carregados sejam mantidos na cache por mais tempo e, assim, seja possível obter melhor aproveitamento da localidade temporal. A escolha do tamanho de bloco permite que dados adjacentes a um dado recentemente acessado também sejam carregados para a cache e assim se obtém melhor aproveitamento da localidade espacial.

Tamanho da Palavra de Dados

Os primeiros itens a serem determinados em um projeto de memória cache são o tamanho da palavra de dados e a capacidade. A palavra é a menor unidade que pode ser lida ou escrita na cache; normalmente corresponde ao mesmo tamanho das instruções do processador e varia de 1 a 8 bytes, ou de 8 a 64 bits. A capacidade depende do espaço disponível no CI e do desempenho desejado ou estimado.

Após a definição do tamanho da palavra e da capacidade desejada na cache é determinado o tamanho da linha ou bloco. Um bloco é formado por uma ou mais palavras, não possuindo um limite máximo específico, visto que quanto mais palavras existirem no bloco mais pode-se aproveitar a localidade espacial. Na prática, por questões de desempenho, utiliza-se um bloco de 8 a 128 bytes, pois a utilização de blocos muito grandes aumenta muito o tráfego de dados de/para a memória principal. Este aumento de tráfego se deve ao fato de que, quando uma palavra do bloco deve ser carregada para a cache para que seja entregue ao processador, todas as outras palavras do bloco também serão carregadas.

Por último é calculado o número de blocos da cache e é definida sua organização, que pode ser de mapeamento direto ou associativa (conceito descrito a seguir). Esta sequência

de escolhas para os parâmetros de projeto foi utilizada para facilitar o entendimento e não é necessariamente a usada em projetos reais. Abaixo é apresentada a fórmula para obtenção da capacidade (em bytes) de uma cache com mapeamento direto.

$$Capacidade = N^{o}BytesNaPalavra * N^{o}PalavrasNoBloco * N^{o}BlocosNaCache$$

Endereçamento da Cache

O endereço de uma palavra de memória é dividido em três campos para se efetuar o mapeamento de um bloco na cache: *offset* do bloco, índice e *tag* (etiqueta) [10], conforme ilustra a Figura 2.2.

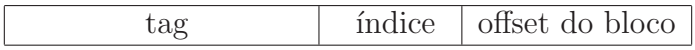

Figura 2.2: Endereçamento da cache

O *offset* do bloco é a parte do endereço destinada a selecionar uma palavra dentre as várias no bloco. O índice e a *tag* formam o endereço do bloco propriamente dito; o índice é usado para selecionar um bloco (ou conjunto) na cache, enquanto que a *tag* é usada para determinar se o bloco desejado está presente ou não na cache.

Associatividade

Um conceito relacionado ao número de blocos existentes na cache e à forma com que estes são dispostos é o conceito de *associatividade*. A cache é de *mapeamento direto* quando uma determinada palavra só pode ser inserida em um único bloco (índice) na cache. Chama-se de cache *totalmente associativa* a cache na qual uma palavra pode ser inserida em qualquer bloco, independentemente de índice. Quando uma palavra pode ser inserida em mais de um bloco com o mesmo índice, diz-se que a cache é associativa por conjunto de N elementos ou vias (N*-way set associative*). Os formatos de mapeamento mais utilizados são o mapeamento direto, o associativo de 2 vias (ou binário) e o associativo de 4 vias (ou quaternário).

Posto de outra forma, cada posição na cache apontada por um índice pode ser considerada como um conjunto com um certo número de elementos. Na cache com mapeamento

direto, os conjuntos da cache possuem apenas um elemento, enquanto que numa cache totalmente associativa todos os blocos são elementos de um único conjunto (não há índice). Em caches com associatividade entre a unária e a total, os conjuntos possuem um número intermediário de elementos, tipicamente dois ou quatro elementos em cada conjunto, para caches com associatividade binária e quaternária, respectivamente.

Modelo dos 3Cs

Sempre que o processador precisa de uma instrução para ser executada, ou necessita de um dado para executar uma instrução, ele faz uma solicitação à memória. Quando estas informações são encontradas em algum nível da hierarquia de memória, dizemos que houve um "acerto na cache" (*hit*), caso contrário dizemos que houve uma "falta na cache" (*miss*).

Uma falta na cache resulta em um acesso ao próximo nível na hierarquia e pode provocar a parada do processador devido à grande latência da memória principal. As faltas são classificadas em três categorias, conhecidas como o modelo qualitativo dos "três Cs" (3Cs), como segue:

- Compulsória - faltas compulsórias são as que ocorrem quando um dado é referenciado pela primeira vez. Ocorrem com maior freqüência durante o início da execução dos programas, fase em que as características da cache não contribuirão para a redução de faltas. A utilização de busca antecipada de dados, bem como de blocos com muitas palavras, podem reduzir a ocorrência das faltas compulsórias.

- Capacidade - faltas por capacidade são as que ocorrem devido ao tamanho limitado da cache. Este tipo de falta depende principalmente da relação entre o tamanho da cache e o conjunto de trabalho (*working set*) do programa que está em execução, sendo fracamente relacionadas ao tamanho do bloco ou ao grau de associatividade da cache. Estas faltas podem ser reduzidas com a utilização de algumas das técnicas de aumento de desempenho de caches que são apresentadas no Capítulo 4.

- Conflito - faltas por conflito são as que ocorrem quando duas referências são mapeadas para um mesmo bloco na cache, a mais nova substituindo a referência mais

antiga. Faltas por conflito podem ser consideravelmente reduzidas com a utilização de associatividade e de algumas das técnicas discutidas no Capítulo 4.

Políticas de Escrita

Além da leitura de instruções e dados da cache, os processadores também efetuam atualizações de dados na cache, e conseqüentemente na memória. As escritas na cache são orientadas por dois modelos de projeto, cada um com suas vantagens, como segue:

- *Write-through* - os dados são gravados em todos os níveis de cache e na memória principal. O *write-through* apresenta como benefícios a fácil implementação e a simplificação do controle da coerência de dados nos sistemas multiprocessados. O tráfego de escritas entre uma cache *write-through* e o próximo nível da hierarquia de memória pode ser intenso.

- *Write-back* - os dados são atualizados apenas na cache. O bloco modificado será gravado no próximo nível da hierarquia somente quando for substituído na cache. A vantagem do *write-back* é a redução no tráfego de dados entre a cache e os demais níveis da hierarquia, ao custo de circuitos de controle mais complexos que os de uma cache *write-through*.

2.4 Processadores Segmentados

A *segmentação* (*pipelining*) é uma técnica de implementação de processadores que aumenta a taxa de execução de instruções por unidade de tempo (*throughput*) através da diminuição do ciclo de relógio do processador e do paralelismo na execução das instruções [10]. A redução do ciclo de relógio é obtida com a *segmentação* dos circuitos eletrônicos do processador, que passam a formar estágios independentes e permitem a execução parcial de várias instruções em um mesmo ciclo de processamento. A execução concorrente de instruções é chamada de paralelismo em nível de instrução (*Instruction Level Parallelism* - ILP).

Os estágios de um processador segmentado básico são interligados em série, formando uma espécie de canal (*pipe*) por onde, a cada ciclo de relógio do processador, uma instrução

pode entrar por uma extremidade, ao iniciar sua execução, avançar de estágio, ou sair pela outra extremidade, ao finalizar sua execução.

O modelo clássico de processador segmentado em 5 estágios é o processador MIPS R3000 [10]. Os estágios são:

1. busca (*Instruction Fetch* - IF)
2. decodificação e acesso aos registradores (*Instruction Decode and register fetch* - ID)
3. execução (EX)
4. acesso à memória (MEM)
5. escrita nos registradores (*register Write Back* - WB)

Em condições ideais, a divisão do processador em estágios com períodos de execução perfeitamente equilibrados reduziria o tempo gasto para executar uma instrução na proporção direta do número de estágios. Caso fossem criados cinco estágios em um processador que gasta um ciclo de relógio para executar uma determinada instrução, ele passaria a executar esta mesma instrução em um ciclo cinco vezes mais curto do que o anterior. Na prática isto não é possível porque os estágios não são perfeitamente equilibrados, ficando a duração do ciclo do processador limitada pelo estágio mais lento. Cada estágio acrescenta uma certa quantia de atraso (*overhead*) na execução das instruções devido aos registradores de segmento e lógica de controle em cada estágio. Além disso, o modelo seqüencial de execução pode ser violado em uma implementação segmentada porque a execução das instruções não é atômica. A segmentação introduz *dependências de dados* —um resultado não pode ser usado pela instrução subseqüente até que seja gravado nos registradores— e *dependências de controle* —as instruções que seguem um desvio devem ser anuladas caso o desvio seja tomado.

2.5 Processadores Superescalares

Uma importante métrica de desempenho para a avaliação de sistemas computacionais é o número médio de ciclos por instrução (CPI), que é a razão entre o número de ciclos de

relógio dispendidos para a execução de um programa, e o número de instruções executadas do programa: $CPI = numCiclos/numInstr$.

O CPI ideal que pode ser alcançado em um processador segmentado é igual a $1,0$ pois no máximo uma instrução é emitida, e conseqüentemente completada, a cada ciclo de relógio. Para se obter melhor desempenho e explorar o paralelismo em nível de instrução, várias instruções devem ser emitidas e completadas em paralelo, permitindo a obtenção de CPIs menores do que $1,0$.

Os processadores superescalares se caracterizam por emitir simultaneamente mais de uma instrução por ciclo de relógio [10]. Além desta característica principal, existem inúmeras outras que podem fazer parte destes processadores dependendo do uso para o qual são projetados, tais como a existência de várias unidades funcionais, o escalonamento estático ou dinâmico de instruções e a execução especulativa de instruções.

Os avanços citados acima propiciam aos processadores superescalares a capacidade de eliminar conflitos por recursos e no uso de dados, resolver dependências de controle e esconder a latência das operações da memória com a execução de instruções independentes, que isoladamente ou em conjunto permitem aumentar a taxa de execução de instruções.

Tipicamente, um processador superescalar causa maior demanda à hierarquia de memória porque efetua um ou mais acessos à memória por ciclo de relógio, visto que a cada quatro instruções (em média) há uma instrução de acesso a dados (*load* ou *store*) [10].

CAPÍTULO 3

AMBIENTE DE SIMULAÇÃO

Neste capítulo são apresentados as ferramentas e os métodos utilizados durante o desenvolvimento da pesquisa, como o simulador, a escolha de parâmetros de simulação, os programas de teste e a coleta e tratamento dos resultados.

3.1 SimpleScalar

O SimpleScalar (SS) versão 3.0d (original) é um conjunto de simuladores altamente configuráveis que permite a criação de modelos de arquiteturas de processadores e hierarquias de memória, bem como a avaliação de desempenho detalhada destes modelos com a execução de programas reais ou de teste [21, 3, 1].

As simulações neste trabalho foram executadas sobre versões modificadas do simulador *sim-outorder*, o qual suporta emissão e execução de instruções fora de ordem (*out-of-order*) e permite a medição precisa do tempo de execução.

Modelo do Sistema de Memória no SimpleScalar

O SimpleScalar permite a configuração de uma hierarquia de memória de três níveis: cache de primeiro nível (L1), cache de segundo nível (L2) e memória principal. As caches L1 e L2 podem ser separadas para dados e instruções, ou unificadas. Normalmente são utilizadas caches L1 individuais e uma cache L2 unificada.

Burger *et al* [2] apresentam uma versão modificada do SimpleScalar original que inclui como características a livre modelagem da hierarquia de memória, o uso de caches não-bloqueantes com número configurável de registradores e a simulação detalhada da tradução de endereços. Neste trabalho utilizou-se uma hierarquia de memória composta apenas por uma cache de dados de primeiro nível (com ao menos um circuito especializado) e memória principal (DRAM); preferiu-se desenvolver as novas funcionalidades a partir da

versão original (3.0d) do simulador, pois esta vem sendo amplamente utilizada em trabalhos de análise de desempenho, e é a que mais facilita efetuar modificações no modelo do sistema de memória.

3.2 Parâmetros de Simulação

Como parte da metodologia de teste dos modelos de memória, os parâmetros de simulação foram divididos em dois grupos: parâmetros variáveis e fixos. Os parâmetros variáveis são aqueles que, a cada experimento, podem ter o seu valor alterado de forma individual e estão normalmente relacionados aos objetos de estudo da pesquisa, tais como tamanho da memória cache, tamanho do bloco e grau de associatividade da cache. Os parâmetros fixos são os que não são alterados individualmente no decorrer da pesquisa, e normalmente são modificados todos em conjunto. São exemplos a largura do processador, o número de unidades funcionais no processador, o número de registros na Register Update Unit (RUU) e o número de registros na Load Store Queue (LSQ).

3.2.1 Parâmetros Variáveis

Os parâmetros variáveis foram escolhidos de maneira a permitir uma investigação realista das memórias cache, pois representam as principais características a serem determinadas no momento do projeto de um novo sistema. A Tabela 3.1 exibe a relação dos valores escolhidos para os parâmetros variáveis.

Parâmetros	Valores
Tamanho da cache (KB)	1, 2, 4, 8 e 16
Largura do bloco (bytes)	8, 16, 32 e 64
Associatividade da cache	1
Modelo de CPU	1, 2, 4 e 8

Tabela 3.1: Valores dos parâmetros variáveis

Cada Modelo de CPU é formado por uma série de parâmetros fixos, como escalaridade e tamanho da LSQ. Suas configurações são apresentadas na próxima seção.

3.2.2 Parâmetros Fixos

Para a execução dos experimentos foram definidos quatro modelos de processadores (CPUs), cada um definido pela tríade ⟨*escalaridade, tamanho da LSQ, tamanho da RUU*⟩. Por *escalaridade* entende-se a largura dos estágios de busca, de decodificação, de emissão e de retirada (*write-back*) do *pipeline*, bem como o número de unidades lógicas e aritméticas (ALUs) de inteiros e ponto-flutuante. A *LSQ* é a fila que armazena as referências à memória que estão em execução, e a *RUU* é o *buffer* de reordenação de instruções [24], responsável pela renomeação de registradores e por garantir que as instruções sejam completadas em ordem. A Tabela 3.2 apresenta os parâmetros para os quatro modelos de CPU utilizados.

Modelo de CPU	1	2	4	8
Escalaridade	1	2	4	8
Tamanho da LSQ	4	8	16	32
Tamanho da RUU	8	16	32	64

Tabela 3.2: Modelos de CPU

O tamanho das palavras de dados e instruções utilizados é de 32 bits. A cache de dados de primeiro nível é diretamente mapeada, possui duas portas de comunicação para suportar até uma leitura e uma escrita por ciclo de processamento e possui latência de acesso de 1 ciclo. O barramento de dados entre a cache e a memória possui largura de 8 bytes (duas palavras) e a latência de acesso à memória é de 18 ciclos para a primeira referência a um bloco, mais 2 ciclos por par de palavras subseqüentes do bloco. Por exemplo, a carga de bloco de 32 bytes custa 24 ciclos = 18+2+2+2. Em todas as simulações a cache de instruções utilizada é de 32 KB, com capacidade relativamente grande para que seu desempenho não interfira na operação da cache de dados, e possui latência de acesso de 1 ciclo.

A listagem a seguir apresenta um exemplo de configuração utilizando-se um processador de escalaridade 2:

```
tamanho das palavras                4 (bytes)(instr/dados)  [fixo]
execução em ordem               false                       [-issue:inorder]
latência de acesso a memória     18 2 (<prim.> <próx.>)     [-mem:lat]
largura do barramento de mem.       8 (bytes)               [-mem:width]
```

tamanho da RUU	16	[-ruu:size]
tamanho da LSQ	8	[-lsq:size]
tamanho da fila de busca	2 (instr.)	[-fetch:ifqsize]
largura da decodificação	2 (instr./ciclo)	[-decode:width]
largura da emissão	2 (instr./ciclo)	[-issue:width]
número de ALU's de inteiros	2	[-res:ialu]
número de mult/div inteiros	1	[-res:imult]
número de portas entre CPU e L1	2	[-res:memport]
número de ALU's de ponto flut.	2	[-res:fpalu]
número de mult/div ponto flut.	1	[-res:fpmult]
largura da retirada	2 (instr./ciclo)	[-commit:width]

3.2.3 Escolha dos Parâmetros Fixos

O embasamento adquirido com o estudo de pesquisas anteriores, também relacionadas à avaliação de memórias cache, facilitou a escolha dos valores dos parâmetros para a execução das primeiras simulações, principalmente em se tratando de parâmetros variáveis e tendo em vista o direcionamento aos sistemas embarcados. Contudo, em vários trabalhos nota-se que alguns parâmetros fixos são simplesmente baseados em valores comumente utilizados, sem serem apresentadas as razões para a sua escolha.

Em [26] os autores simularam um processador com largura 8, RUU de 256 registros, LSQ de 128 registros e cache L1 com 8 portas, para avaliar a tolerância à latência das instruções *load*. O grande número de registradores na RUU e na LSQ utilizados exerce uma acentuada pressão sobre o sistema de memória, mas não é praticável devido ao alto custo de implementação. Como contra exemplo, é possível citar o processador Pentium 4 [9], o qual utiliza 128 registradores na RUU e 72 registradores na LSQ (48 para *loads* e 24 para *stores*). No entanto, como esta pesquisa foi direcionada para os sistemas embarcados, optou-se por utilizar valores de parâmetros mais modestos.

Para descobrir a influência dos parâmetros fixos no desempenho das memórias cache e então escolher os valores a serem utilizados durante a pesquisa, foram efetuadas várias séries de experimentos alterando-se os parâmetros fixos.

Estas simulações foram realizadas utilizando-se uma cache de instruções de 32 KB, uma

cache de dados não-bloqueante de 2 KB diretamente mapeada com 4 MSHRs (*Miss Status Holding Register*, para atender a 4 faltas concorrentes), processadores com escalaridades 1, 2, 4 e 8, blocos com larguras 16, 32 e 64 bytes e executou-se o programa de teste *anagram* que vem incluído no pacote do simulador. Testou-se RUUs com 8, 16, 32 e 64 registradores e LSQs com 2, 4, 8, 16 e 32 registradores.

Analisando-se os resultados dos experimentos constatou-se um baixo nível de ocupação da RUU e da LSQ para as configurações e programas de teste utilizados, razão pela qual se decidiu utilizar registradores RUU e LSQ relativamente pequenos e condizentes com uma implementação de baixo custo.

A Tabela 3.3 apresenta resultados dos experimentos para uma CPU com escalaridade 2 e blocos de 16 bytes. As colunas com a abreviatura "ocup." representam os números médios de registros ocupados, e as com o termo "cheia" indicam a percentagem do tempo em que os conjuntos de registradores estão totalmente preenchidos. Para esta largura de CPU optou-se por utilizar uma RUU de 16 registros e uma LSQ de 8 registros, que, como mostra a segunda linha da Tabela 3.3, é a combinação mais eficaz.

Exp.	Tam. RUU	Tam. LSQ	RUU ocup.	RUU cheia (%)	LSQ ocup.	LSQ cheia (%)	Taxa faltas	IPC	CPI
1	16	4	9,59	11	2,25	20,5	8,44	1,06	0,9436
2	16	8	11,18	32	2,62	2,7	8,45	1,08	0,9265
3	32	8	15,41	7	3,55	7,9	8,23	1,17	0,8549
4	32	16	16,18	17	3,74	0,8	8,26	1,19	0,8430
5	64	16	21,29	4	4,78	0,8	8,11	1,21	0,8234
6	64	32	21,54	4	4,94	0,6	8,12	1,22	0,8227

Tabela 3.3: Experimentos para escolha dos parâmetros fixos

Para a configuração escolhida, a ocupação média da RUU é de 11,18 registros, e esta permanece cheia por 32% do tempo, enquanto a LSQ apresenta uma ocupação média de apenas 2,62 registros e fica cheia menos de 3% do tempo. Pode-se notar que o aumento no número de registradores na RUU e na LSQ, a partir dos valores selecionados, produz pouco acréscimo nos seus níveis de ocupação.

3.3 CommBench

O CommBench é um conjunto de programas de teste composto por núcleos de programas que realizam computação intensa e refletem aplicações que são comuns em processadores de rede [30]. O CommBench é formado por programas de processamento de cabeçalho (*Header-Processing Applications* - HPA) e de conteúdo (*Payload Processing Applications* - PPA), os quais, dentre outras funções, são responsáveis por efetuar operações de encaminhamento de pacotes ou análise e transformação de conteúdo.

A avaliação de desempenho foi realizada com a execução de seis programas de teste do conjunto. São eles: CAST-128 - algoritmo de criptografia (enc) e descriptografia (dec) aritmética; DRR - algoritmo de escalonamento (*Deficit Round Robin*); FRAG - programa de fragmentação de pacotes IP; JPEG - algoritmo de compressão (enc) e descompressão (dec) de imagens. A Tabela 3.4 mostra o número de instruções executadas, o número de referências à memória, a taxa de referências e o número médio de instruções entre cada desvio dos programas quando executados no simulador SimpleScalar para qualquer configuração.

Característica	Nr.inst.	Nr.refs.	Refs.	Nr.loads	Entre dsv.	Tipo
Programa / unid.	×1000	×1000	%	×1000		
DRR	212.806	126.091	59,25	104.323	4,83	HPA
FRAG	42.965	10.308	23,99	6.403	5,07	HPA
CAST dec	137.786	41.333	29,99	28.662	11,44	PPA
CAST enc	137.786	41.333	29,99	28.662	11,44	PPA
JPEG dec	219.683	67.340	30,65	42.204	10,00	PPA
JPEG enc	302.894	81.619	26,94	54.398	8,24	PPA

Tabela 3.4: Características dos programas de teste no SimpleScalar

3.4 Execução dos Experimentos

Os experimentos foram executados nas servidoras do Departamento de Informática (DINF) da Universidade Federal do Paraná (UFPR). Das 6 máquinas utilizadas, conseguiu-se um total de 20 processadores disponíveis, visto que elas têm mais de um processador e possuem configurações diferentes.

Em virtude do número de combinações de parâmetros e de modelos de cache simula-

dos, foram coletados 63.165 resultados, os quais consumiram um total de 6.514 horas de processamento, equivalentes a 271 dias em apenas um processador.

Dada a grande quantidade de testes a serem realizados, foram criados *scripts bash* para a submissão automatizada dos experimentos. Os *scripts* foram separados em pastas nomeadas por programa de teste e em cada pasta foi criado um *script* para cada modelo de cache a ser simulado. O apêndice B contém um exemplo de *script* para execução do modelo de cache MSHR com o programa FRAG.

3.5 Coleta e Tratamento dos Resultados

Para auxiliar o manuseio dos dados coletados nos experimentos foi criada uma base de dados (BD) relacional alocada no sistema de gerenciamento de bases de dados (SGBD) PostgresSQL [16]. Desta base de dados é possível extrair os resultados para posterior análise e geração de gráficos ou tabelas utilizando-se a linguagem SQL.

Os dados dos experimentos foram preparados para inserção na base de dados com a execução de um *script awk* sobre os resultados de cada simulação. No apêndice B é mostrado o *script awk* que prepara os dados para inserção na base de dados e o *script* que define e permite criar a estrutura da base de dados utilizada. Ainda, no apêndice B, é apresentado um roteiro para preparação do ambiente de simulação.

CAPÍTULO 4

MODELOS DE CACHE E SEU DESEMPENHO

Neste capítulo são apresentadas algumas das técnicas que possibilitam melhoria no desempenho das memórias cache, juntamente com seus pontos fortes e/ou fracos. Para os modelos avaliados são detalhadas as formas de implementação e são apresentados e discutidos seus desempenhos individuais. Por fim são comentados trabalhos relacionados. A ordem de exposição dos modelos é pela cronologia de publicação.

Cabe lembrar que as implementações dos modelos de cache no simulador SimpleScalar visam permitir a contabilização do tempo de execução (medido em ciclos), das taxas de acerto e de faltas, bem como de outros parâmetros que permitem a avaliação e comparação de desempenho entre os modelos. Portanto, o formato das estruturas e/ou registradores aqui implementados podem se apresentar ligeiramente diferentes das propostas originais. Os modelos descritos consideram uma hierarquia de memória com uma cache primária (L1), uma ou mais estruturas auxiliares e memória principal (DRAM).

4.1 Cache Não-Bloqueante

A técnica de cache não-bloqueante (*Lockup-Free Cache*), apresentada por Kroft em 1981 [15], reduz a degradação de desempenho causada pelas paradas do processador quando ocorre uma falta na cache. A técnica consiste na adição de registradores e circuitos de controle que permitem que o processador continue executando instruções mesmo com a ocorrência de faltas na cache.

Estes registradores, chamados de MSHRs (*Miss Status Holding Registers*), guardam o estado da falta e mantém todas as informações necessárias para a sua resolução, tais como o endereço de memória do dado solicitado e o número da unidade funcional do processador que aguarda pelo dado, além de vários campos de controle que indicam se o MSHR está em uso e se seu conteúdo é válido, por exemplo.

Em uma cache não-bloqueante uma falta na cache é registrada em um MSHR enquanto os dados faltantes são buscados da memória. Durante a busca, outras referências a dados podem ser servidas diretamente pela cache.

Em um determinado sistema de memória, o número de MSHRs determina quantas faltas pendentes podem ser atendidas simultaneamente pela cache. O processador somente pára a execução de instruções se não existir um MSHR disponível para atender uma falta, ou ainda se ocorrerem múltiplos acessos para um mesmo bloco da cache. Considerando questões de custo e ganho de eficiência, Kroft concluiu que o ideal é utilizar quatro MSHRs e que isto aumenta em cerca de 10% o custo do projeto da cache (dados de 1981).

Em 1994, Farkas e Jouppi descreveram vários métodos para implementação de caches não bloqueantes [5]. Alguns destes métodos resolvem o problema das paradas do processador devido a múltiplos acessos para um mesmo bloco da cache. Mostraram também que uma redução de 2 a 10 vezes no número de ciclos de processador parado pode ser conseguida utilizando-se 4 MSHRs. O aumento potencial de paralelismo só pode ser conseguido com processadores superescalares, capazes de emitir/executar mais do que uma instrução por ciclo.

4.1.1 Implementação do MSHR

O modelo de MSHR implementado segue a forma proposta por Kroft, incluindo-se as características indicadas por Farkas e Jouppi para permitir que o processador continue em execução mesmo na ocorrência de acessos a endereços próximos que mapeiam para um mesmo bloco. A partir deste ponto do trabalho o conjunto de registradores MSHR será referenciado simplesmente como MSHR.

No apêndice A são listadas as estruturas, em linguagem de programação C, utilizadas para modelar os MSHRs. Cada registrador do MSHR é composto pelos campos *addr* (endereço), *tag* e *ready* (pronto), bem como pelo indicador *valid* (registro válido e ocupado). A Figura 4.1 mostra um conjunto de MSHRs e a localização destes no modelo da hierarquia de memória. Abaixo são descritos o comportamento e a forma de contabilização de ciclos e taxas de faltas/acertos do MSHR.

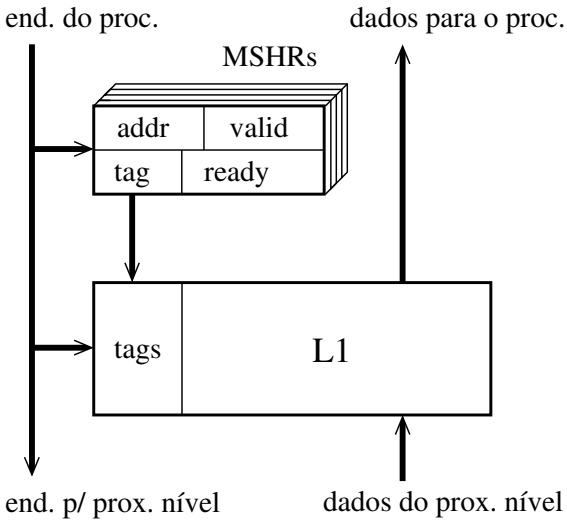

Figura 4.1: Estrutura da cache não-bloqueante.

1. Quando ocorre acerto na L1:

   1.1. o dado é entregue diretamente da L1 para o processador;

   1.2. é computado custo de acesso de 1 ciclo;

   1.3. é contabilizado um acerto na L1.

2. Quando ocorre falta na L1, falta no MSHR (o endereço do bloco não corresponde a nenhum dos endereços carregados nos registradores do MSHR que estão em uso) e existe registrador livre no MSHR (indicador de válido com valor zero):

   2.1. um registrador do MSHR é alocado (indicador de válido marcado com valor um);

   2.2. o endereço de memória que provocou a falta é salvo no campo *addr*;

   2.3. o valor do campo *tag* é determinado e preenchido;

   2.4. o próximo nível da hierarquia de memória é consultado e retorna o número de ciclos necessários para o atendimento da falta (custo do acesso);

   2.5. o ciclo em que a falta será atendida (ciclo atual mais custo para atendimento da falta) é gravado no campo *ready*;

   2.6. o custo de acesso é retornado para o processador e o registro do MSHR permanecerá em uso até que o ciclo de processamento alcance o valor do campo *ready*;

2.7. é contabilizada uma falta na L1 e no MSHR.

3. Quando ocorre falta na L1, falta no MSHR e não existe registrador livre no MSHR:

   3.1. o processador pára a execução de novas instruções (passa para o próximo ciclo de processamento) até que exista um registrador livre no MSHR.

4. Quando ocorre falta na L1 e acerto no MSHR (o endereço do bloco acessado corresponde a um endereço armazenado em um registro do MSHR; pode ocorrer em palavras diferentes no mesmo bloco):

   4.1. o custo de acesso é contabilizado em função do ciclo em que o bloco estará disponível (*ready*), descontado o ciclo do acesso atual (tempo restante para a falta ser atendida), e este é retornado ao processador;

   4.2. é contabilizada uma falta na L1 e um acerto no MSHR.

5. Quando o MSHR indicar bloco pronto e o ciclo atual de processamento for igual ao valor do campo *ready*:

   5.1. o bloco é carregado na L1;

   5.2. o registrador do MSHR é marcado como inválido (indicador de válido marcado com valor zero).

Para configurar o número de MSHRs a serem utilizados em uma dada simulação foi adicionado o parâmetro *-cache:mshr_dl1* ao simulador. Com este parâmetro devem ser passados dois argumentos: o primeiro é o nome que será usado para identificar o conjunto de registradores no arquivo de resultados e o segundo é o número de registradores desejado. Por exemplo, a nova linha no arquivo de configurações do simulador mostrada abaixo define uma estrutura identificada pelo nome “MSHR” e que contém 4 registradores.

```
-cache:mshr_dl1           MSHR:4
```

4.1.2 Comparação com o SimpleScalar Original

O SimpleScalar original (SS_orig - versão 3.0d) não possui controle sobre o número de faltas a serem atendidas simultaneamente pela hierarquia de memória. Seu modelo de interface entre as caches L1 e L2/MEM pressupõe que existe um número ilimitado de MSHRs e, portanto, o sistema de memória pode suportar infinitas faltas concorrentes. No SS_orig, sempre que ocorre uma falta em um nível da hierarquia (L1), o bloco é imediatamente carregado para este nível e o custo para carregar o bloco do nível inferior (L2) é computado e gravado no campo de controle do nível que originou a falta (L1). Esta é uma hipótese razoável para o projeto de processadores superescalares agressivos, uma das aplicações do SimpleScalar. No entanto, para processadores menos ambiciosos, como os de aplicações embarcadas, é desejável um modelo mais realístico de interface de memória.

Como o modelo MSHR implementado sobre o SS_orig permite definir o número de faltas simultâneas a serem atendidas pela hierarquia de memória, decidiu-se utilizá-lo como base da avaliação de desempenho entre modelos no restante da pesquisa. A seguir é apresentado o estudo comparativo entre o SS_orig e a versão com MSHR.

4.1.3 Resultados com Número Finito de MSHRs

Após efetuar a adaptação do simulador para usar os MSHRs, uma série de experimentos foram executados para comparar os resultados da limitação do número de faltas concorrentes em relação à versão original do simulador (SS_orig). A Figura 4.2 mostra as alterações no número de instruções completadas por ciclo (IPC) ao variar-se os tamanhos da cache e do bloco para duas versões extremas de largura de processador; uma corresponde a um processador de emissão/execução simples (largura 1) e a outra a um processador superescalar que pode emitir/executar oito instruções por ciclo (largura 8). Os valores exibidos correspondem à média dos resultados dos seis programas de teste.

Os efeitos de uma cache não-bloqueante em um processador estreito são mínimos, pois em um ciclo qualquer existe no máximo um *load* ou *store* na interface de memória. Como esperado, os resultados para este processador mostram pouca diferença no desempenho entre o SS_orig e o modelo com MSHR. Para blocos grandes (64 bytes), o MSHR adicionou

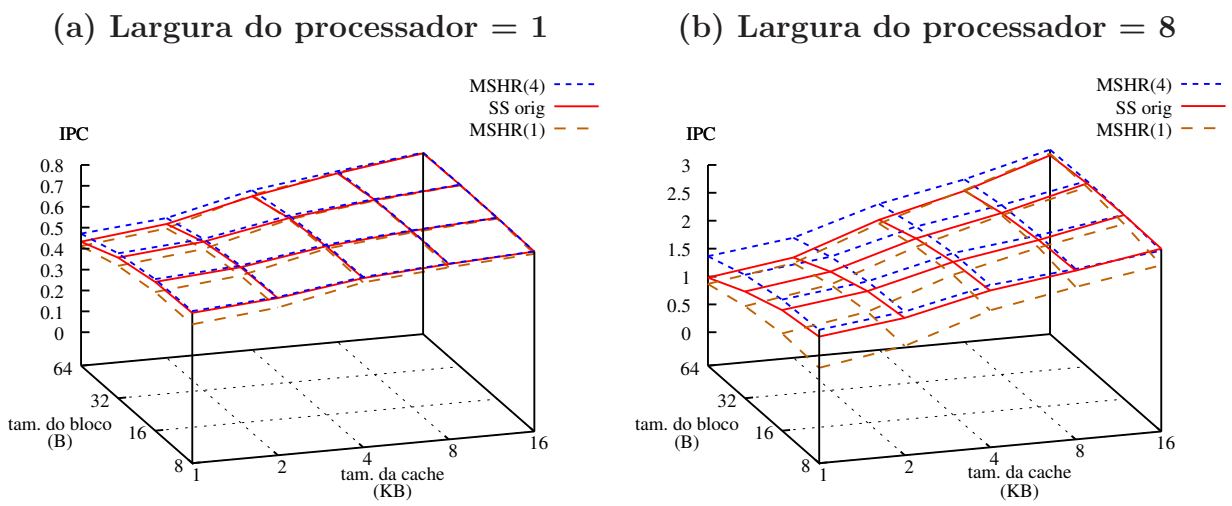

Figura 4.2: SS Original *versus* MSHR

um grau limitado de associatividade na cache diretamente mapeada (L1) uma vez que referências secundárias para uma falta pendente são todas atendidas pelo MSHR, o que implica em latências e taxas de faltas menores. As curvas do gráfico para 1 MSHR mostram uma dininuição no IPC quando comparado com um número ilimitado de MSHRs. Isto era esperado por causa da contabilização mais precisa da latência e da taxa de faltas.

O processador com maior largura impõe uma alta demanda ao sistema de memória, uma vez que é possível executar, concorrentemente, até oito referências à memória. Como pode ser visto na Figura 4.2 (b), para caches com muitos blocos (8 bytes/bloco) existe pouca diferença entre o modelo original e o com 4 MSHRs. Para caches pequenas com poucos blocos (1 KB=16x64 bytes/bloco) o número de acertos secundários no MSHR é suficiente para melhorar o desempenho do modelo com 4 MSHRs em cerca de 0,5 IPC. Este ganho no desempenho é explicado pelo pequeno grau de associatividade acrescido pelos MSHRs e pelo fato de que o bloco, que será substituído no momento em que uma falta terminar de ser atendida pelo MSHR, permanece na L1 durante o período de resolução da falta, possibilitando que ocorram mais acertos em seu conteúdo. Como era esperado para esta largura de processador, um único MSHR piora o desempenho em todas as combinações de tamanhos de bloco e cache.

As Figuras 4.3 e 4.4 ilustram com mais detalhes os resultados do modelo MSHR com 1, 2, 4, e 16 registradores se comparado ao SS_orig. A Figura 4.3 mostra o resultado

para um processador com escalaridade 2 enquanto a Figura 4.4 para um processador com escalaridade 4, ambos com blocos de 32 bytes de tamanho. No lado esquerdo das figuras são exibidas as curvas das taxas de faltas (média dos seis programas de teste). Nelas pode-se perceber quanto o limitado grau de associatividade adicionado pelos blocos do MSHR beneficia a taxa de faltas, em cerca de 10% para caches de 1 e 2 KB na configuração da Figura 4.3 e em 18% para os mesmos tamanhos de cache na configuração da Figura 4.4. Nota-se também que, independente do número de registradores utilizados no MSHR, a taxa de faltas permanece praticamente inalterada; isto demonstra que os padrões de referência do conjunto de programas utilizados não permite tirar proveito da capacidade adicionada pelo uso de vários MSHRs. Ainda, para caches de 16 KB, as baixas taxas de faltas indicam que este tamanho de cache comporta praticamente todo o conjunto de dados dos programas.

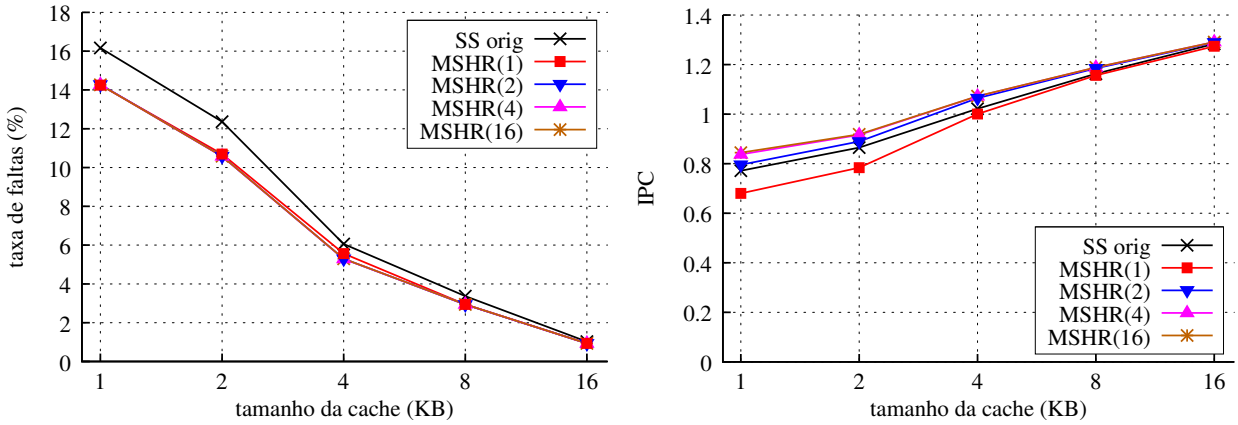

Figura 4.3: MSHR *versus* SS Original, larg.=2, t_b=32

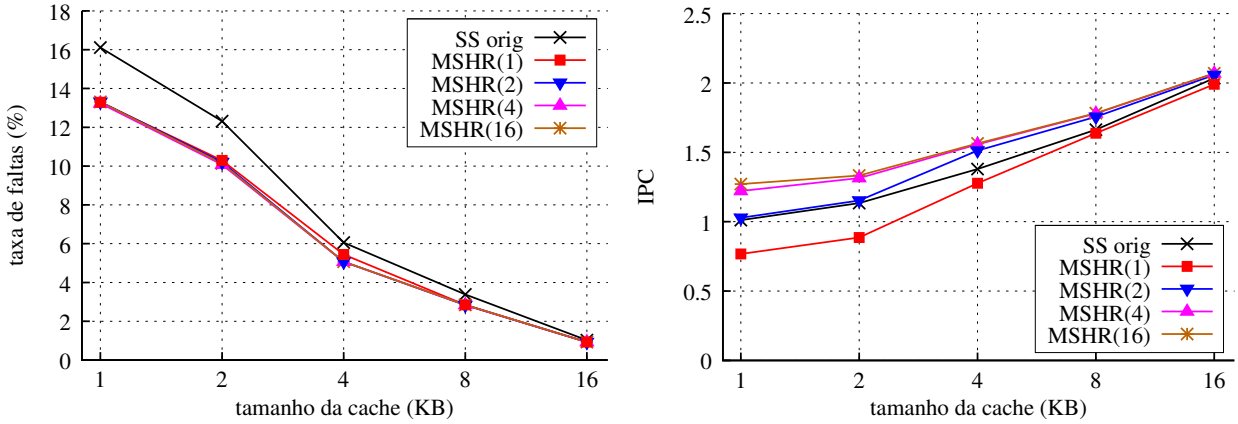

Figura 4.4: MSHR *versus* SS Original, larg.=4, t_b=32

Por outro lado, a variação no número de MSHRs influencia diretamente o número de

instruções completadas por ciclo (IPC). Como pode ser visto no lado direito das Figuras 4.3 e 4.4, a utilização de apenas 1 MSHR provoca a serialização no acesso à memória e consequentemente uma redução no IPC, se comparado ao SS_orig. Contudo, com 2 ou mais MSHRs, o IPC se iguala ou supera o do SS_orig, mas não é necessário utilizar mais do que 4 MSHRs, visto que o ganho obtido com a utilização de 16 registradores é muito pequeno, nas condições destes experimentos.

Todos os modelos a seguir são implementados e executados sobre o modelo com 4 MSHRs e os seus resultados são apresentados para processadores de escalaridades 2 e/ou 4 e tamanhos de bloco 32 bytes (a menos que seja dito o contrário), pois estes tamanhos correspondem às configurações mais utilizadas nos sistemas embarcados.

4.2 Cache de Faltas

A cache de faltas foi apresentada em 1990 por Jouppi e consiste em uma pequena cache totalmente associativa que contém de 2 a 5 linhas de dados [13]. A cache de faltas tem a função de adicionar um pequeno grau de associatividade às caches diretamente mapeadas de primeiro nível, devendo ser inserida no barramento de comunicação entre as caches L1 e L2/MEM como mostra a Figura 4.5.

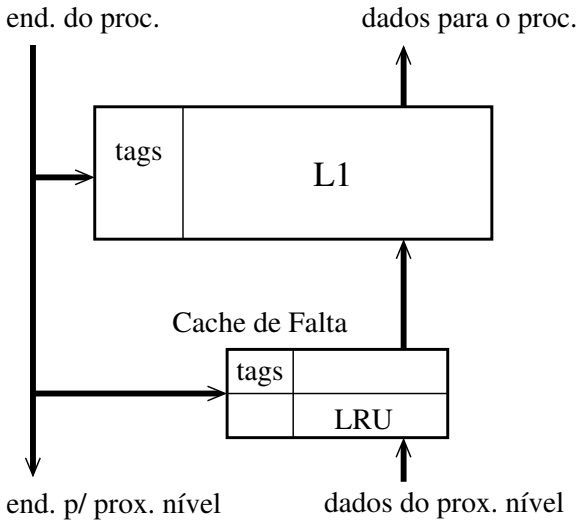

Figura 4.5: Cache de faltas.

Esta cache funciona em conjunto com a L1 e opera da seguinte forma:

1. quando ocorre uma falta na L1 e na cache de faltas, o dado carregado do nível inferior da hierarquia de memória é alocado em ambas as caches;

2. quando todas as linhas da cache de faltas estão ocupadas, será substituída a linha de dados menos recentemente utilizada (*Least Recently Used* - LRU);

3. quando ocorre uma falta na L1 seguida de um acerto na cache de faltas, o dado da cache de faltas é copiado para a L1, e a cópia custa um ciclo adicional.

A cache de faltas reduz o número de acessos dispendiosos aos níveis mais baixos na hierarquia de memória. Note que as pesquisas nas etiquetas ocorrem paralelamente na L1 e na cache de faltas.

A cache de faltas mostrou-se eficiente em remover faltas de conflito em caches L1 com mapeamento direto, visto que causa somente um ciclo de penalidade para o caso de falta na L1 com acerto na cache de faltas. Como ponto negativo, desperdiça uma ou mais de suas linhas ao manter uma cópia do(s) dado(s) recentemente transferido(s) para a L1.

Este modelo não foi implementado neste estudo porque a cache de vítima apresenta melhores resultados.

4.3 Cache de Vítima

A cache de vítima (*Victim Cache* - VC) é uma extensão da cache de falta [13]. Na cache de vítima aplica-se uma política distinta para a carga dos dados a fim de se evitar a duplicação de linhas que ocorre entre a L1 e a cache de faltas.

Somente são carregados na VC blocos que foram expurgados da L1, e que são as vítimas da substituição nas faltas. Isso evita a duplicação de dados entre a VC e a L1 e torna a VC mais eficiente na remoção de faltas de conflito do que a cache de faltas.

Em [13] é demonstrado que uma VC pode eliminar cerca de 20% das faltas de dados de uma cache L1 de 4 Kbytes com mapeamento direto. A VC adiciona um pequeno grau de associatividade em uma cache de mapeamento direto, permitindo eliminar custosas faltas de conflito, sem aumentar significativamente seu tamanho e complexidade. Contudo, existe um ponto negativo: quando uma referência acerta em um bloco na VC, este bloco, além de

ser entregue para o processador, é encaminhado para a L1, e existindo uma nova vítima da L1, esta será carregada na VC. A troca de blocos entre a VC e a L1 implica em um atraso de, ao menos, um ciclo adicional para efetivar a operação[1]. Em [6], Fisk e Bahar mencionam o atraso adicional como uma desvantagem da VC mas não apresentam nenhum dado para dar suporte à reclamação.

4.3.1 Implementação da VC

A modelagem da VC aqui implementada segue a versão original proposta por Jouppi. Para a implementação da VC utilizou-se a estrutura de dados de cache já definida no simulador, sendo necessário contudo adicionar os novos comportamentos ao código do simulador. A Figura 4.6 ilustra a VC, e em seguida são apresentadas as características do modelo:

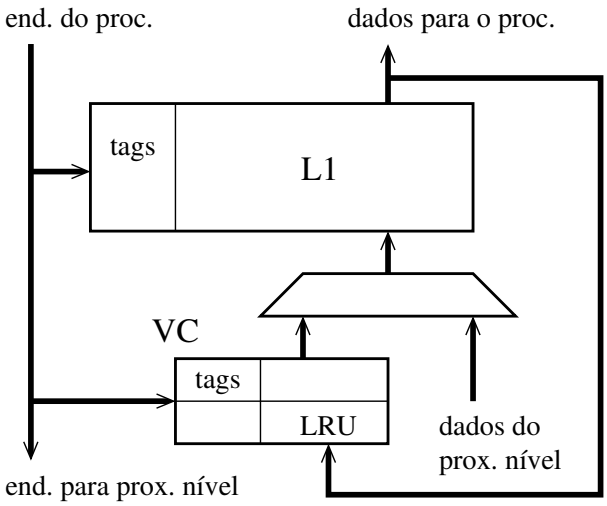

Figura 4.6: Cache de Vítima.

1. L1 e VC são acessadas em paralelo;

2. todos os blocos expurgados da L1 são alocados na VC utilizando a política LRU para as trocas;

3. um acerto na L1 custa 1 ciclo;

[1]Não foi encontrado nenhum comentário sobre o ciclo adicional em [13].

4. uma falta na L1 com acerto na VC custa 1 ciclo para entregar a palavra para a CPU, mais 1 ciclo para efetuar a troca (*swap*) dos blocos (durante a troca de blocos a L1 permanece bloqueada para novos acessos);

5. falta em ambas custa um acesso à memória.

Para configurar a VC no simulador foram adicionados dois parâmetros. O parâmetro *-cache:dcv* deve receber cinco argumentos: o nome que será usado para identificar a VC no arquivo de resultados, o número de blocos por conjunto na VC (deve ser sempre 1), a largura do bloco da cache, o número de conjuntos na VC (corresponde ao número de blocos desejado, pois a VC é totalmente associativa) e a política de substituição dos blocos (deve ser sempre l = LRU). O parâmetro *-cache:dcvlat* deve receber o valor da latência do acesso à VC. O exemplo dos parâmetros no arquivo de configurações do simulador define uma cache de vítimas chamada "DCV" com blocos de largura de 8 bytes, 4 blocos e latência 1.

```
-cache:dcv              DCV:1:8:4:l
-cache:dcvlat                   1
```

4.3.2 Resultados da VC

Os resultados de simulação para a cache de vítima são mostrados na Figura 4.7, com um processador de largura dois, tamanho de bloco de 32 bytes, e esta é comparada ao modelo base que é equipado com 4 MSHRs ⟨MSHR(4)⟩.

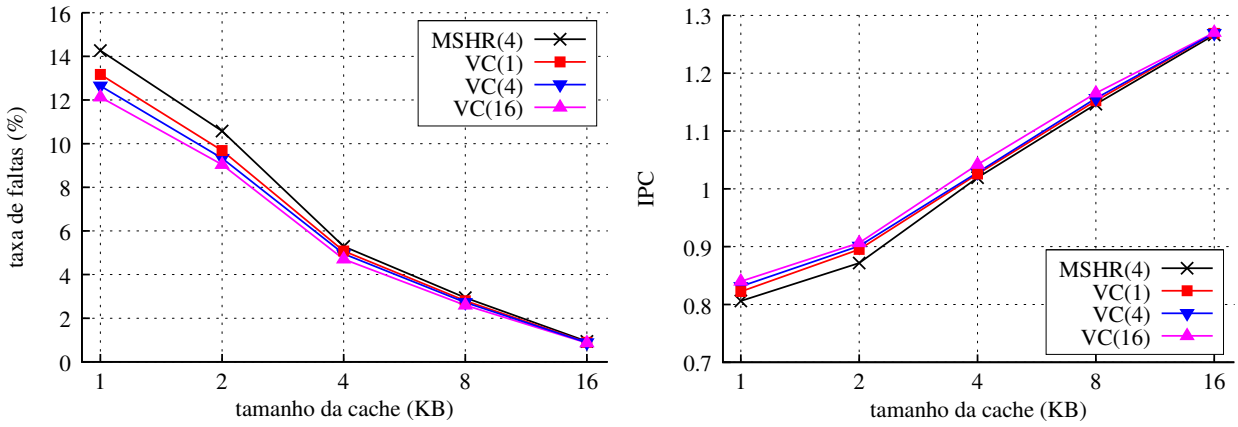

Figura 4.7: Cache de Vítima *versus* MSHR, larg.=2, t_b=32

Na esquerda são mostradas as taxas de faltas médias e na direita os IPCs médios, considerando-se os seis programas de teste. Para uma cache de 1 KB, a adição de uma cache de vítima com 16 blocos ⟨VC(16)⟩ melhora a taxa de faltas em 11%, enquanto que com uma VC de apenas um bloco ⟨VC(1)⟩ a melhoria é de 6%. A VC adiciona capacidade (mais blocos) e associatividade a caches L1 de mapeamento direto, sendo os ganhos mais evidentes para caches pequenas.

Os ganhos em relação ao IPC correspondem diretamente aos das taxas de faltas. Uma VC com 4 blocos ⟨VC(4)⟩, comparada ao modelo base, melhora o IPC em 2% e 2,6% para caches de 1 e 2 Kbytes, respectivamente.

4.4 Influência do Tamanho dos Blocos da Cache

As escolhas da capacidade, do tamanho da palavra de dados, do tamanho do bloco e da organização da cache são fundamentais para o desempenho do sistema de memória. Nesta seção é discutida a influência do tamanho do bloco no desempenho de uma cache L1 com mapeamento direto. São apresentados resultados de simulação para os modelos de memória do SimpleScalar original e das versões com MSHR e Cache de Vítima.

A Figura 4.8 ilustra os resultados das médias de taxa de faltas (esq.) e IPC (dir.) para os seis programas de teste utilizados, com processador de largura 2 e cache de 2 KB, variando-se o tamanho do bloco entre 8 e 64 bytes. Nota-se que as menores taxas de faltas são obtidas com o maior tamanho de bloco (64 bytes) e que, por outro lado, os maiores IPCs são obtidos com o bloco de menor tamanho (8 bytes).

A taxa de faltas melhora com a utilização de blocos grandes porque estes aproveitam melhor a localidade espacial dos programas. No entanto, o IPC diminui porque estes blocos possuem grande latência de preenchimento e são substituídos frequentemente devido ao seu menor número para uma dada capacidade. O contrário ocorre para blocos pequenos, nos quais a diminuição na localidade espacial provoca um aumento na taxa de faltas mas a menor latência de preenchimento melhora o IPC. Como mencionado nas seções 4.1.3 e 4.3.2, o modelo com VC(4) obtém a melhor taxa de faltas e também o melhor IPC quando comparado ao modelo MSHR(4) que, por sua vez, supera o SS original.

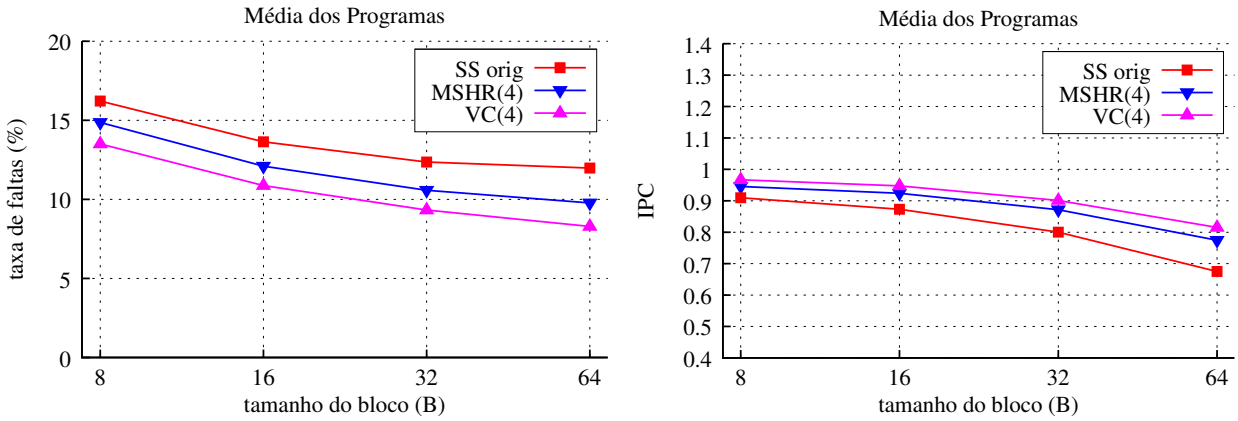

Figura 4.8: Influência do Tamanho do Bloco - todos os programas, larg.=2, t_c=2KB

As Figuras 4.9 e 4.10 apresentam resultados médios para os programas de teste separados entre os que processam cabeçalhos (HPA) e os que processam conteúdos (PPA), respectivamente.

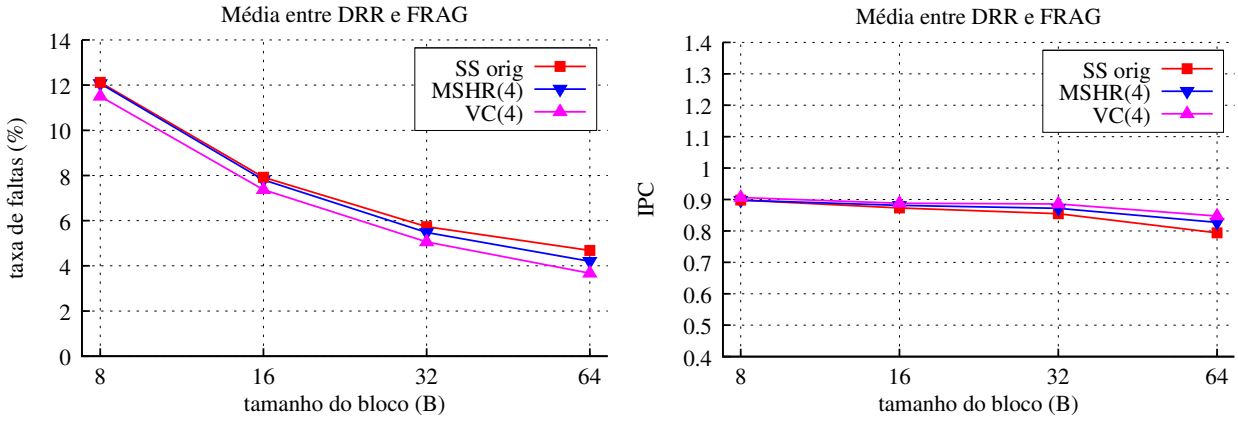

Figura 4.9: Influência do Tamanho do Bloco - programas HPA, larg.=2, t_c=2KB

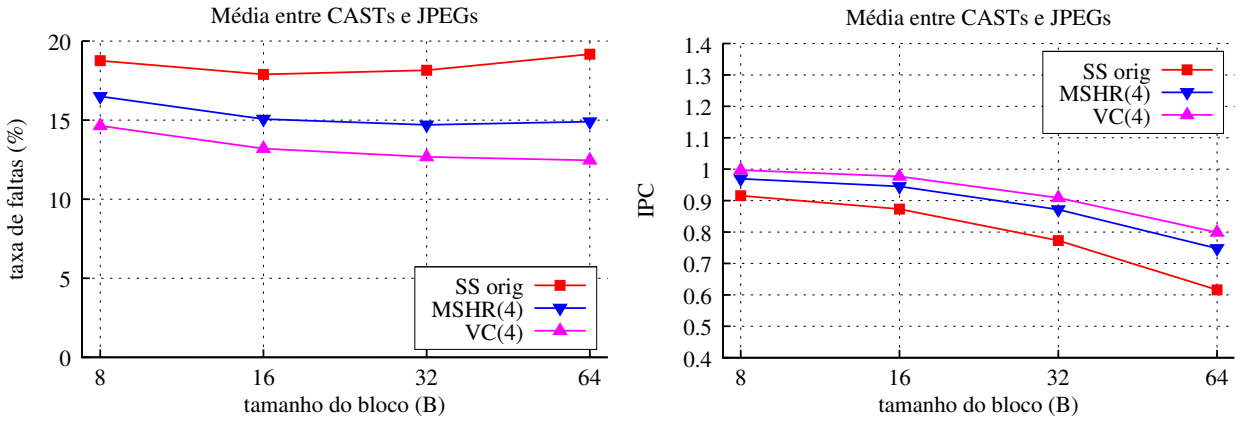

Figura 4.10: Influência do Tamanho do Bloco - programas PPA, larg.=2, t_c=2KB

Para os programas do tipo HPA (fig. 4.9) as menores taxas de faltas são obtidas utilizando-

se blocos de 64 bytes, embora com este tamanho de bloco o IPC se torne de 6% a 8% menor do que nos outros tamanhos. Ainda, o IPC permanece praticamente invariável para tamanhos de bloco iguais a 8, 16 e 32 bytes, enquanto a taxa de faltas é reduzida em aproximadamente 30% a cada duplicação no tamanho.

Para os programas do tipo PPA (fig. 4.10) as menores taxas de faltas ocorrem para blocos de tamanhos médios (16 e 32 bytes) e os maiores IPCs são obtidos com blocos de 8 bytes porque estes possuem menor latência de preenchimento. Estes programas percorrem todo o corpo dos pacotes e se beneficiam de localidade espacial. Por outro lado, ao percorrer os pacotes, seus conjuntos de trabalho são maiores e dispersos, o que é evidenciado pelas taxas de faltas com a VC(4) e com o MSHR(4), reduzidas graças à associatividade e aos blocos adicionais providos pelos dois modelos.

Tendo em vista que os tamanhos de bloco de 16 e 32 bytes são os que produzem os resultados com melhor relação taxa de faltas *versus* IPC, os demais resultados deste estudo são apresentados com estes tamanhos de bloco.

4.5 *Stream Buffer*

O *stream buffer* (SB) é um mecanismo que realiza a busca antecipada de dados da L2 para a cache de primeiro nível [13]. Este *buffer* consiste em uma fila de um ou mais registros, cada qual contendo um campo de endereço, um indicador de disponível e uma linha de dados semelhante a um bloco da L1. A fila é disposta entre a L1 e a L2 como ilustra a Figura 4.11.

Quando ocorre uma falta na L1 e o bloco solicitado não está na fila, o *stream buffer* inicia a busca do bloco faltante e dos seus sucessores na L2, a partir do endereço que originou a falta. Ao ser disponibilizado, o bloco que provocou a falta é entregue à L1 e os demais permanecem armazenados no *buffer*, permitindo que novas faltas na L1, desde que para blocos subseqüentes, sejam atendidas em apenas um ciclo adicional.

Sempre que um bloco é entregue à L1, a primeira posição da fila é liberada e todos os demais blocos são deslocados de uma posição, liberando a última posição. Neste momento o *buffer* inicia a busca de um novo bloco para preencher a posição vaga.

O *stream buffer* é eficiente na eliminação de faltas de capacidade e faltas compulsórias

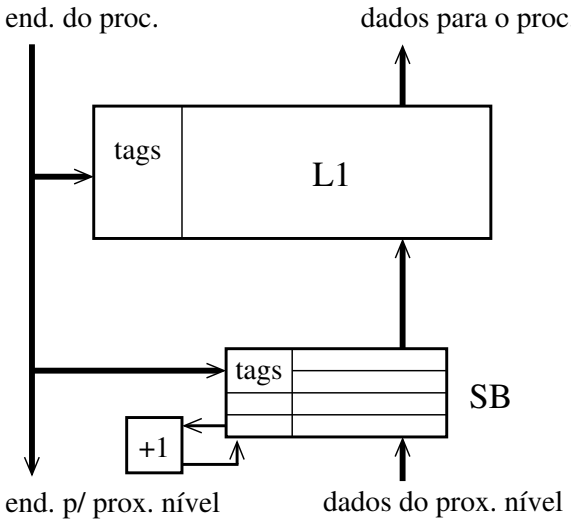

Figura 4.11: *Stream Buffer*.

na cache L1, bem como na remoção de faltas de conflito na cache L1 de instruções. Segundo os dados apresentados em [13], a adição deste *buffer* a uma hierarquia de memória simples, permite remover 72% das faltas na cache L1 de instruções e 25% das faltas na cache L1 de dados utilizando-se um único *buffer* com quatro blocos. Caso seja utilizado um SB associativo de quatro conjuntos com quatro blocos em cada conjunto, 43% das faltas na L1 de dados podem ser eliminadas. No entanto, o autor não faz referência ao aumento do tráfego de dados no barramento que pode ser intensificado dramaticamente em decorrência das buscas antecipadas, as quais nem sempre são úteis.

4.5.1 Implementação do SB

A implementação do SB segue o modelo proposto em [13], exceto pela comparação de *tags* que é realizada em todos os blocos do conjunto. Cada bloco do SB é composto pelos campos *tag*, *status* e *ready* (pronto), bem como por dois ponteiros que indicam os registradores anterior e seguinte (caso existam). Para o caso de SB associativo é agregada uma estrutura de dados adicional que permite selecionar (com política de LRU) qual conjunto deve ser utilizado no caso de uma nova busca. No Apêndice A são listadas as estruturas de dados do SB.

Além dos contadores de faltas e acertos da estrutura principal do SB, foram adicionados

contadores para medir o seu padrão de utilização. A seguir é descrito o comportamento do SB em conjunto com a L1.

1. L1 e SB são acessados em paralelo;
2. uma falta na L1 dispara o início de busca pelo SB;
3. uma falta na L1 com acerto no SB custa 1 ciclo se a palavra estiver no SB, senão custa o tempo restante para carregar o bloco da memória;
4. falta na L1 e falta no SB, para SB com profundidade maior que 1:
   (n = larg. bloco / larg. barramento)
   4.1. o primeiro bloco é entregue com custo da latência do acesso à memória mais a transferência, $(18 + ((n - 1) \cdot 2))$ ciclos;
   4.2. os blocos subseqüentes custam $(n \cdot 2)$ ciclos;
   4.3. todos os blocos de uma seqüência de busca que estão prontos no SB são pesquisados na tentativa de aumentar a taxa de acertos;
5. controle da ocupação do barramento: a requisição ocorre em 1 ciclo e em seguida libera o barramento; após tempo de espera pela resposta, a tranferência do bloco ocupa o barramento por n ciclos.

Note que a simulação do barramento é detalhada. Se o barramento está ocupado, uma transação de memória é atrasada pelo número de ciclos decorridos até a liberação do barramento.

A configuração do SB no simulador é realizada por meio de dois parâmetros. O parâmetro *-streambuf:dsb* deve receber quatro argumentos: o nome que será usado para identificar o SB no arquivo de resultados, o número de blocos por conjunto no SB (profundidade do *buffer*), a largura do bloco da cache e o número de conjuntos no SB (grau de associatividade). O parâmetro *-streambuf:dsblat* deve receber o valor da latência do acesso ao SB. O exemplo abaixo com os parâmetros no arquivo de configurações do simulador configura um SB chamado “DSB” com 4 blocos, sendo que cada bloco possui 8 bytes, associatividade 1 e latência de 1 ciclo.

```
-streambuf:dsb          DSB:4:8:1
-streambuf:dsblat               1
```

4.5.2 Resultados do SB

As *tags* no *stream buffer* simulado são verificadas em todos os blocos e em todas as vias porque isto melhora o tempo de acesso e a taxa de faltas, inclusive para seqüências com larguras de passo variáveis. Em um acerto no SB, a palavra requisitada é encaminhada para o processador e para a L1. Nas figuras mostradas abaixo, a organização dos *buffers* é dada como $\langle$SB(*profundidade*, *associatividade*)$\rangle$, sendo profundidade o número de blocos por conjunto no SB e associatividade o número de conjuntos no SB. A Figura 4.12 mostra a ocupação do barramento para processadores de largura 1 (larg.=1) e 4 (larg.=4) e tamanho de bloco de 32 bytes (t_b=32). Este tamanho de bloco foi escolhido porque causa o maior tráfego no barramento. Para SBs com um único bloco $\langle$SB(1, x)$\rangle$, a ocupação no barramento é aproximadamente 30% maior que a do modelo base. SBs com mais do que quatro blocos tem uma elevadíssima ocupação, que é reduzida com mais associatividade, como pode ser visto para $\langle$SB(4, 1)$\rangle$ e $\langle$SB(4, 4)$\rangle$, por exemplo.

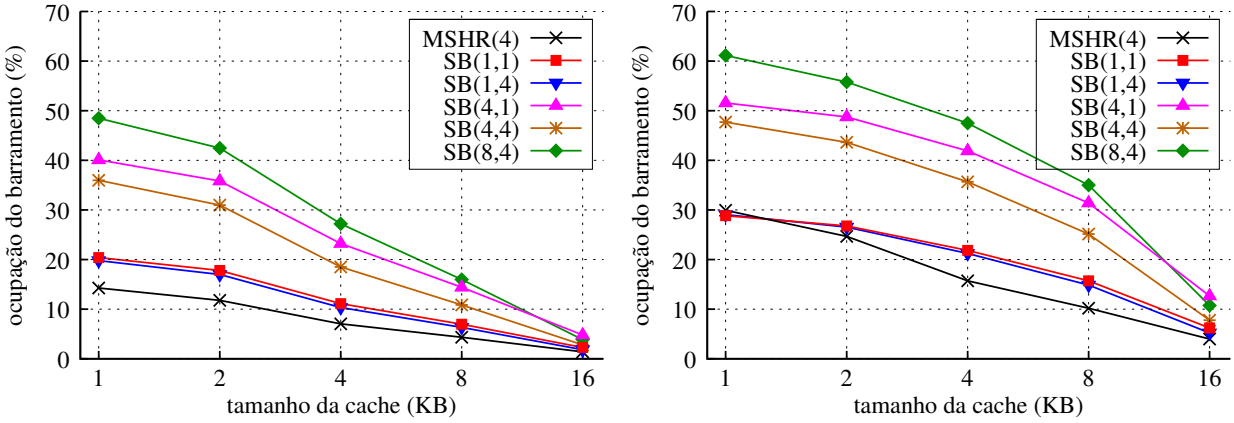

Figura 4.12: SB: ocupação do barramento, larg.=1 (esq.) e larg.=4 (dir.), t_b=32

Em [13] o autor investiga o efeito da taxa de faltas no SB e ignora a ocupação no barramento e o IPC. A Figura 4.13 mostra curvas da taxa de faltas (esquerda) e do IPC (direita) para um processador de largura dois (larg.=2) e tamanho de bloco de 32 bytes.

Os SBs pequenos, $\langle$SB(1, 1)$\rangle$ e $\langle$SB(1, 4)$\rangle$, causam as menores ocupações no barramento

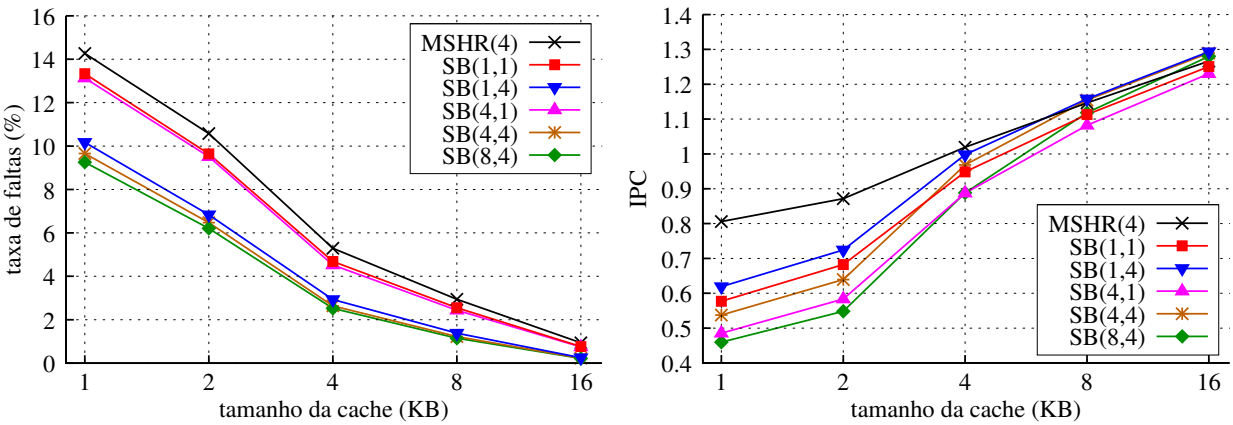

Figura 4.13: *Stream Buffer versus* MSHR, larg.=2, t_b=32

e as maiores taxas de faltas enquanto apresentam os melhores desempenhos, ou talvez, não os piores desempenhos quando comparados ao modelo base. Excluindo as maiores caches, o SB causa uma diminuição no desempenho. Isto pode ser explicado por uma estimativa grosseira: existe uma referência à memória a cada 4 instruções (2 ciclos para largura=2), com mais de uma falta a cada 10 referências; existe uma falta a cada 20 ciclos e cada falta é atendida em 20 ciclos; se o SB provoca buscas adicionais da memória (muitas delas sem aproveitamento) então a ocupação do barramento e o tempo de preenchimento da cache aumentam e, em consequência, o desempenho do processador piora.

Para esta hierarquia de memória e conjunto de programas testados o SB não tem bom desempenho. Ele pode apresentar melhor desempenho em um sistema com barramento mais complexo ou se colocado entre a cache L2 e a memória principal, onde o tráfego no barramento é menos intenso do que entre a cache L1 e a memória principal.

4.6 Cache de Escrita

A cache de escrita (*write cache*) foi proposta por Jouppi em 1993 [14] e consiste em uma pequena cache totalmente associativa capaz de reter e agrupar escritas de dados em seus blocos, diminuindo com isto o tráfego de escritas para os níveis mais baixos da hierarquia de memória.

Como explicado por Jouppi, a cache de escrita opera em conjunto com uma fila de escrita (*write buffer*) ou pode ser associada a esta fila [22], bem como pode ser implementada com

a funcionalidade de cache de vítima. Nos testes efetuados pelo autor, com uma cache de escrita de apenas 8 blocos, ocorre uma redução em 50% no número de escritas para a memória principal. Esta técnica não foi investigada nesta pesquisa.

4.7 Cache de Controle de Poluição

Segundo Walsh e Board [29], a "poluição" ocorre quando blocos pouco referenciados são alocados na cache no lugar de blocos freqüentemente referenciados e esta situação provoca a ocorrência de duas faltas, uma para inserir na cache o bloco pouco utilizado, e outra para recarregar o bloco comumente utilizado que foi retirado da cache. Este comportamento tende a aumentar a ocorrência de faltas por capacidade e faltas por conflito.

Para solucionar o problema da poluição, Walsh e Board apresentaram a cache de controle de poluição (*Pollution Control Caching* - PCC). A PCC é uma pequena cache totalmente associativa que opera em paralelo à cache L1 [29]. Com a PCC, ao contrário de outras técnicas, dados referenciados pela primeira vez não são carregados diretamente na L1, mas sim na PCC. Dados são escritos na L1 se, e somente se, eles já foram encontrados na PCC. Isto evita que dados muito utilizados sejam retirados da L1 na substituição por dados pouco utilizados, reduzindo a probabilidade de ocorrência de poluição. A Figura 4.14 ilustra o modelo da cache PCC em sua versão original.

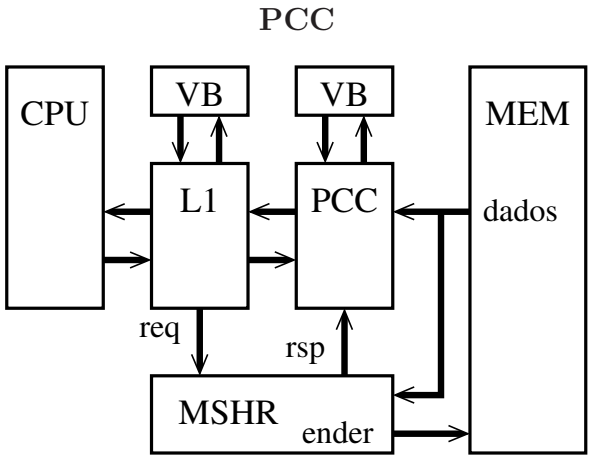

Figura 4.14: Modelo de PCC originalmente proposto

Durante uma operação de leitura, a PCC e a L1 são pesquisadas em paralelo; se um

acerto ocorre na L1, o ciclo é completado normalmente; se o acerto ocorre na PCC, o dado é retornado para o processador e então gravado na L1; uma falta ocorre somente se o dado não é encontrado em nenhuma das caches.

Os resultados apresentados por Walsh e Board mostram que uma cache L1 de 4KB, operando em conjunto com uma PCC e dois *buffers de vítima* (*Victim Buffer* - VB, conforme ilustra a Figura 4.14), um junto à L1 e outro junto à PCC, obtém desempenho semelhante à uma cache L1 de 32KB, demonstrando as vantagens do PCC quanto ao ganho de desempenho e à redução de espaço ocupado pela cache. O *buffer de vítima* é uma pequena modificação da cache de vítima apresentada por Jouppi.

4.7.1 Implementações da Cache de Controle de Poluição

Foram implementados dois modelos distintos de PCC. O primeiro, denominado de PCC1, segue parcialmente o modelo proposto por Walsh, utiliza uma cache de vítima (VC) junto à L1, além da cache de controle de poluição (PCC). Como descrito anteriormente, no modelo proposto em [29] foram utilizados dois *buffers de vítima*, um junto à L1 e outro junto à PCC. O segundo, denominado de *Cache de Vítima e Controle de Poluição* (Pollution Control Victim Cache - PCVC), é um modelo novo (proposto neste trabalho) e utiliza a própria cache de controle de poluição como cache de vítima para a L1 (PCC + VC). A Figura 4.15 ilustra os dois modelos de cache de controle de poluição implementados.

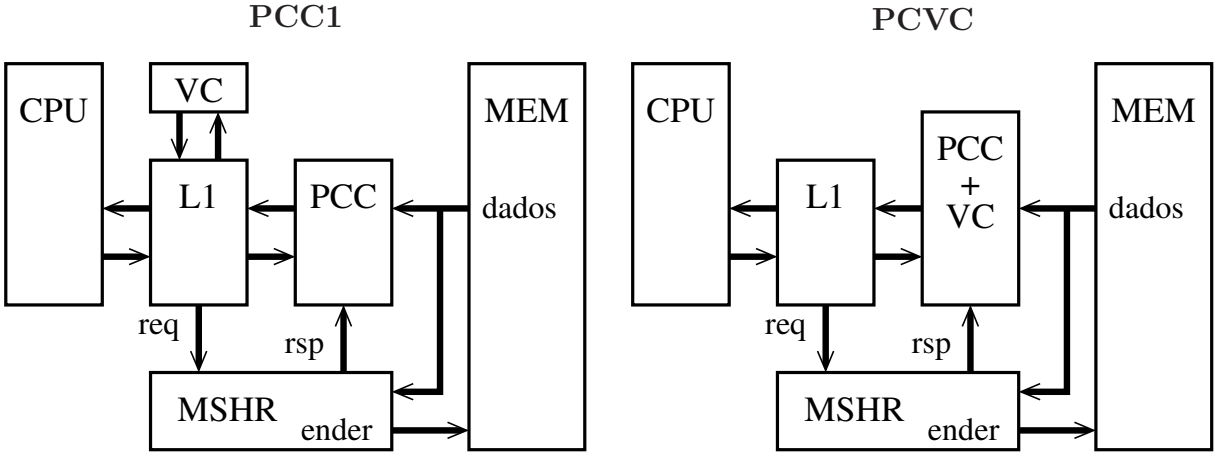

Figura 4.15: Modelo da PCC1 e da PCVC

Para a implementação da estrutura de dados base dos modelos de PCC foi utilizada a

estrutura de memória cache já existente no simulador, sendo necessário adicionar os novos parâmetros de configuração e os comportamentos dos modelos na codificação do simulador.

Para configurar a PCC no simulador foram adicionados dois parâmetros. O parâmetro *-cache:dpcc* deve receber cinco argumentos: o nome que será usado para identificar a PCC no arquivo de resultados, o número de blocos por conjunto na PCC (deve ser sempre igual a 1), a largura do bloco da cache, o número de conjuntos na PCC (número de registradores) e o indicador da política de substituição de blocos a ser utilizada. O parâmetro *-cache:dpcclat* deve receber o valor da latência do acesso à PCC. O exemplo dos parâmetros no arquivo de configurações do simulador define uma PCC chamada "DPCC" com blocos de largura de 8 bytes, com 4 registradores, que utiliza a política de substituição LRU e tem latência de 1 ciclo.

```
-cache:dpcc         DPCC:1:8:4:l
-cache:dpcclat                1
```

Note que os parâmetros são os mesmos para as duas versões de PCC implementadas. No entanto tratan-se de simuladores diferentes. Assim, na primeira versão do modelo PCC, o PCC1, além da configuração da PCC também será necessário configurar os parâmetros da cache de vítima (VC).

Nas próximas subseções são descritos os detalhes de implementação de cada modelo de PCC implementado.

4.7.2 Modelo PCC1

O modelo PCC1 é caracterizado pela adição de uma cache de vítima (VC) junto à L1. Nesta configuração, os blocos que saem da L1 (vítimas da substituição de blocos) são alocados na VC e a pesquisa dos blocos nos acessos à memória ocorre paralelamente nas três caches, L1, VC e PCC. Abaixo segue a descrição do comportamento da PCC1.

1. a pesquisa é realizada em paralelo na L1, na VC e na PCC;

2. um acerto na L1 custa 1 ciclo e nenhum bloco é movimentado;

3. uma falta na L1 com acerto na VC custa 1 ciclo;

   3.1. o bloco da VC é carregado na L1;

   3.2. e o bloco que sai da L1 vai para a VC;

4. uma falta na L1 com falta na VC mas com acerto na PCC custa 1 ciclo;

   4.1. o bloco da PCC é escrito na L1 e é invalidado na PCC;

   4.2. o bloco que sai da L1 vai para a VC;

5. falta em todas as estruturas custa acesso à memória;

   5.1. o bloco é entregue à CPU;

   5.2. e é carregado na PCC;

6. L1, VC e PCC nunca possuem blocos repetidos;

7. a PCC utiliza política de substituição de blocos LRU;

   7.1. blocos pouco utilizados são simplesmente substituídos.

O modelo PCC1 é o que possui a lógica de controle mais complexa dentre os modelos implementados nesta pesquisa, pois tem que efetuar a busca e controlar o fluxo de blocos em três estruturas distintas (L1, VC e PCC).

Neste modelo, além da parametrização da PCC, a configuração da VC é realizada pelos seus próprios parâmetros, como descrito na Seção 4.3.

4.7.3 Resultados da PCC1

A Figura 4.16 apresenta resultados da taxa de faltas e do IPC para diferentes configurações da PCC1 comparada ao modelo base; ambos utilizam processador de largura 2 e blocos de 32 bytes e apresentam valores para a média dos seis programas de teste. Na figura mostrada abaixo, a organização da PCC1 é dada como ⟨PCC1(*número de blocos na PCC*, *número de blocos na VC*)⟩.

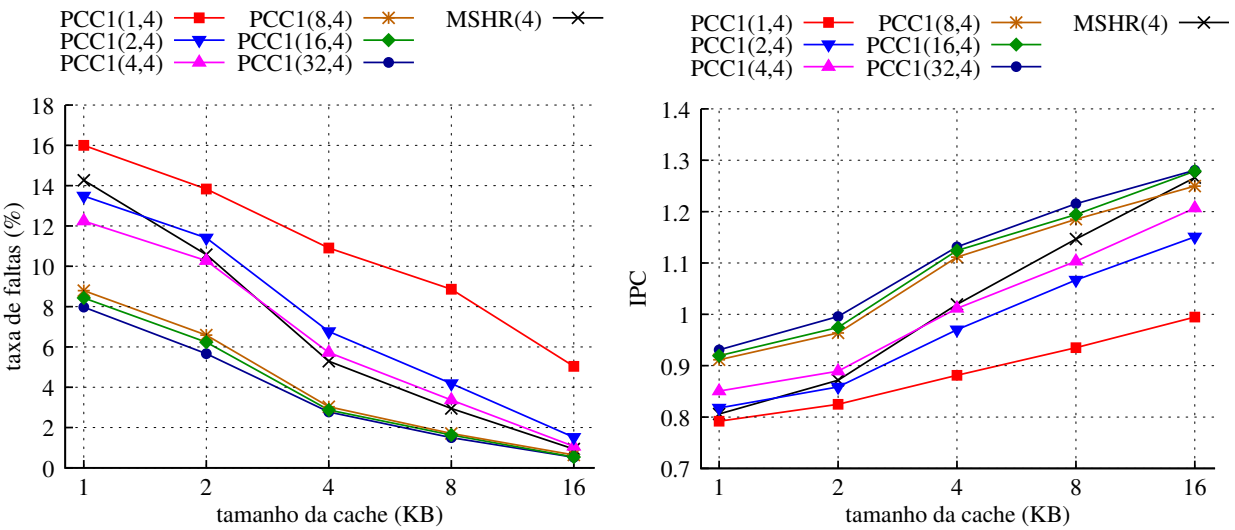

Figura 4.16: PCC1 *versus* MSHR, larg.=2, t_b=32

Como mostra o lado esquerdo da Figura 4.16, se o PCC1 é pequeno (1-4 blocos), ou a taxa de faltas aumenta ou o ganho é pequeno em relação ao MSHR(4). Com PCC1s grandes (8-32 blocos) ocorre uma redução de aproximadamente 50% na taxa de faltas para todos os tamanhos de cache. Para o desempenho global, as altas taxas de faltas dos pequenos PCC1s são refletidas no IPC, como pode ser visto no lado direito da Figura 4.16. Ainda, para os grandes PCC1s, os ganhos no IPC variam de 11% (8 blocos) até 14% (32 blocos) para caches de 1, 2 e 4 KB comparando-se ao modelo base com 4 MSHRs.

Os resultados para a PCC1 apresentam-se muito bons uma vez que uma L1 de 1 KB operando em conjunto com uma PCC1(8,4) de cerca de 0,5 KB (12 blocos de 32 bytes mais barramentos e lógica de controle adicional) produziu melhor desempenho do que uma L1 de 2 KB simples.

4.7.4 Modelo PCVC

A cache de vítima e controle de poluição foi criada durante o desenvolvimento desta pesquisa e surgiu a partir da decisão de agregar os comportamentos da cache de vítima e da cache de controle de poluição em uma única estrutura. A intenção era simplificar a lógica de controle e diminuir o número de blocos utilizados na cache para reduzir seu tamanho e consumo de energia, com a expectativa de manter o desempenho.

Na PCVC, a qual funciona como cache de vítima para a L1, além de ser efetuado o

controle de poluição da L1 pela PCVC, um bloco que sai da L1 é realocado na PCVC. Nesta nova forma de implementação, os registradores da PCVC compartilham as funcionalidades de controle de poluição (PCC) e armazenagem de vítimas (VC), permitindo que sejam utilizados da maneira que for mais eficaz em cada fase da execução. Abaixo segue a descrição do comportamento da PCVC.

1. a pesquisa é realizada em paralelo na L1 e na PCVC;
2. um acerto na L1 custa 1 ciclo e nenhum bloco é movimentado;
3. uma falta na L1 com acerto na PCVC custa 1 ciclo;
    3.1. o bloco da PCVC é escrito na L1 e é invalidado na PCVC;
    3.2. e o bloco que sai da L1 retorna para a PCVC;
4. falta em ambas estruturas custa acesso à memória;
    4.1. o bloco é entregue à CPU;
    4.2. e é carregado na PCVC;
5. L1 e PCVC nunca possuem blocos repetidos;
6. a PCVC utiliza política de substituição de blocos LRU;
    6.1. blocos pouco utilizados são simplesmente substituídos.

A subseção a seguir mostra os resultados de simulação da PCVC comparada ao modelo base e no Capítulo 5.3 é apresentada uma comparação dos desempenhos dos modelos PCC1 e PCVC.

4.7.5 Resultados da PCVC

A Figura 4.17 apresenta resultados da média dos seis programas de teste da taxa de faltas e do IPC para diferentes configurações da PCVC, comparadas ao modelo base com 4 MSHRs. Ambos são executados sobre um processador de largura 2 e blocos de 32 bytes. Na

figura mostrada abaixo, a organização da PCVC é dada como ⟨PCVC(*número de blocos na PCVC*)⟩.

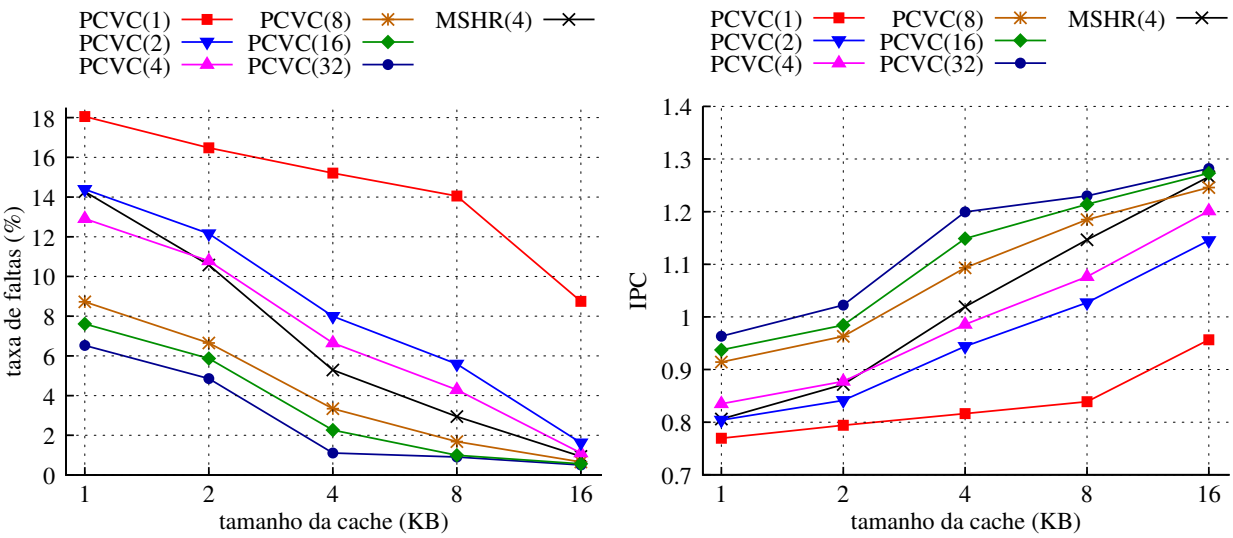

Figura 4.17: PCVC *versus* MSHR, larg.=2, t_b=32

O comportamento dos menores PCVCs (1-4 blocos) é um pouco pior do que o dos menores PCC1s, como mostra a Figura 4.17. Esta diferença é facilmente percebida comparando-se as curvas do PCC1(1,4) com as do PCVC(1) e ocorre porque no modelo PCC1 são utilizados 4 blocos na VC nas nossas simulações.

Os ganhos nas taxas de faltas dos grandes PCVCs variam de acordo com seus tamanhos: para PCVCs de 8, 16 e 32 blocos os ganhos são, respectivamente, de 39%, 47% e 54% em uma cache de 1 KB e de 37%, 50% e 79% em uma cache de 4 KB, comparando-se com o modelo base de 4 MSHRs.

Na PCVC, uma vez que o PCC também atua como uma cache de vítima, o IPC cresce com o número de registros, como mostra o lado direito da Figura 4.17. Para uma cache pequena de 1 KB os ganhos da PCVC de 8, 16 e 32 blocos são de 13%, 16% e 19%, respectivamente.

4.8 Trabalhos Relacionados

Um trabalho com alguma similaridade ao discutido aqui foi apresentado em [25]. Os autores investigam os efeitos dos parâmetros de projeto (organização, associatividade, capacidade e

política de substituição) sobre o desempenho da cache. No entanto, é medida somente a taxa de faltas usando a versão mais básica para simulação de caches do simulador SimpleScalar [3], sobre os programas de teste de propósito geral do pacote SPEC CPU2000.

No que segue são discutidos outros trabalhos relacionados ao desenvolvido e apresentado aqui.

Cache de Fluxo Não-Temporal

Em 1996, Rivers e Davidson propuseram a cache de fluxo não-temporal (*Non-Temporal Streaming* (NTS) *Cache*), um *buffer* totalmente associativo, colocado em paralelo a caches convencionais de mapeamento direto de primeiro nível, e responsável por alocar blocos de memória que apresentam padrão de referência não-temporal [19] [20].

O padrão de referência não-temporal é produzido, por exemplo, em acessos a variáveis do tipo *array*, os quais apresentam grande localidade espacial e pouca ou nenhuma localidade temporal. Um bloco é caracterizado como "não-temporal" (NT) quando nenhuma das suas palavras é referenciada mais do que uma vez durante o período em que o bloco permanece na cache L1. Este comportamento provoca níveis consideráveis de conflitos e poluição na cache pois provoca a retirada de blocos freqüentemente utilizados.

Além dos blocos de memória que constituem a cache NTS, são adicionados novos campos e lógica de controle junto aos blocos da L1 para detectar e indicar o padrão de referência não-temporal e cada bloco da L2 tem um indicador para marcar seus blocos como temporais ou não-temporais. O padrão de referência é detectado enquanto o bloco permanece na L1; quando o bloco deixa a L1, ele é marcado na L2 com o valor da temporalidade; em um próximo acesso a um bloco previamente marcado como NT, este será alocado na cache NTS, caso contrário irá para a L1.

Resultados apresentados no estudo mostram que a cache NTS propicia ganhos de desempenho e de espaço em comparação a caches de mapeamento direto convencionais. Para alguns programas numéricos testados, uma cache NTS de 9KB (8KB dos blocos + 1KB dos novos campos e lógica de controle) apresentou desempenho semelhante ao de uma cache convencional de 16KB. Este aperfeiçoamento de cache exige que sejam adicionados indicadores

de blocos não-temporais em todos os blocos da estrutura de segundo nível da hierarquia de memória do sistema (L2/MEM). Em razão da sua complexidade, este modelo de cache não foi investigado neste estudo, que é focado em estruturas pequenas para sistemas embarcados.

Cache Intercalada Baseada na Localidade

Em 1997, Rivers *et al* descreveram a cache intercalada baseada na localidade (*Locality-Based Interleaved Cache* - LBIC), uma cache que explora o padrão de referências na memória para melhorar a vazão e a latência da busca de dados na cache, bem como para economizar espaço na construção da cache em comparação às caches multi-portas [18].

A LBIC é formada por uma cache multi-bancos e por registradores de várias portas ligados a cada banco e interligados ao processador por uma malha de barramentos de dados e endereços (*crossbar*). Esta organização permite que a cache atenda a mais de um acesso simultâneo por ciclo de processamento, inclusive para uma mesma linha da cache.

Embora este modelo de cache possa fornecer um ganho de desempenho no acesso à memória, sua implementação é demasiado complexa para sistemas embarcados. Por este motivo o modelo foi excluído deste estudo.

Fila de Não-Críticos

A fila de não-críticos (*Non-Critical Buffer* - NCB), apresentada por Fisk e Bahar em 1999, é uma pequena fila totalmente associativa que é acessada em paralelo à cache L1 e possui funcionamento semelhante ao de uma cache de vítima [6]. A diferença está na lógica de preenchimento das linhas, a qual determina de forma dinâmica, em tempo de execução, se os dados devem ser alocados na L1 (dados críticos) ou na fila de não-críticos.

A lógica de preenchimento de linhas foi desenvolvida com base no estudo dos *loads* tolerantes à latência efetuado por Srinivasan e Lebeck em 1998 [26]. Neste estudo os autores mostraram que "de 1% a 62% das cargas devem ser completadas em apenas um ciclo de relógio enquanto de 5% a 98% devem completar em até 8 ciclos, dependendo da configuração do processador". Entre os métodos analisados para o cálculo da criticalidade o que apresentou melhor resultado foi o que mede a quantidade de instruções dependentes. Segundo

os autores “uma falta de leitura (*load*) é considerada crítica se ela possui mais do que um certo número de instruções dependentes enquanto espera para ser atendida”.

Seguindo esta linha de raciocínio, em 2000, Rakvic *et al* dividiram as leituras de dados em vitais e não-vitais [17]. Nesta pesquisa os autores constataram que 25% das cargas são vitais para o desempenho do processador, devendo ser atendidas em 1 ciclo de relógio, e que os 75% restantes são não-vitais e portanto podem sofrer algum atraso (de 1 a 2 ciclos de relógio) sem que haja uma degradação significativa no desempenho do processador.

A fila de não-críticos apresentou resultados melhores que os conseguidos com caches de associatividade 2 e com a cache de vítima, obtendo ganhos de desempenho acima de 4% no número de instruções processadas por ciclo de relógio do processador (*Instructions Per Cycle* - IPC), para alguns dos programas de teste analisados. Esta técnica também não foi investigada devido ao seu custo e complexidade.

CAPÍTULO 5

ANÁLISE DE RESULTADOS

Neste capítulo são apresentados os resultados individuais de desempenho dos programas de teste utilizados (Seção 5.1), é mostrado o padrão de referências no Stream Buffer (Seção 5.2), é realizada uma comparação dos desempenhos dos modelos de cache PCC1 e PCVC (Seção 5.3), bem como é apresentada a composição da taxa de acertos total (Seção 5.4), e o tempo de execução para os modelos testados (Seção 5.5).

5.1 Resultados Individuais dos Programas de Teste

Nesta seção são apresentados os resultados de simulação para os seis programas de teste utilizados. Os dados descrevem o comportamento de cada modelo de cache avaliado frente aos diferentes padrões de acesso à memória. Todas as curvas exibidas nos gráficos são referentes aos modelos MSHR(4), VC(4), SB(1,4), PCC1(32,4) e PCVC(32), todos executados sobre um processador de largura 2 com blocos de 32 bytes. Para tornar o texto mais claro, os modelos serão citados somente por suas siglas, ficando implícitos os seus tamanhos.

Os programas DRR e FRAG possuem como característica em comum o processamento de dados de cabeçalho de pacotes; o primeiro realiza o escalonamento de pacotes para seu tratamento enquanto o segundo fragmenta os pacotes. Seus desempenhos são apresentados nas Figuras 5.1 e 5.2.

Para o programa DRR, os modelos VC, PCC1 e PCVC propiciam ganhos que variam de aproximadamente 4% até 13% no IPC para caches pequenas (1 e 2 KB), enquanto o modelo SB produz ganho semelhante para as caches grandes (8 e 16 KB), como mostra o lado direito da Figura 5.1. Ainda para o mesmo programa, a VC causa pouca redução na taxa de faltas, enquanto o SB, a PCC1 e a PCVC reduziram em cerca de 20% a 33% a taxa de faltas para caches de 1, 2 e 4 KB. Analisando-se o desempenho global com o DRR pode-se dizer que este programa é mais beneficiado pelo aumento na capacidade do que pelo

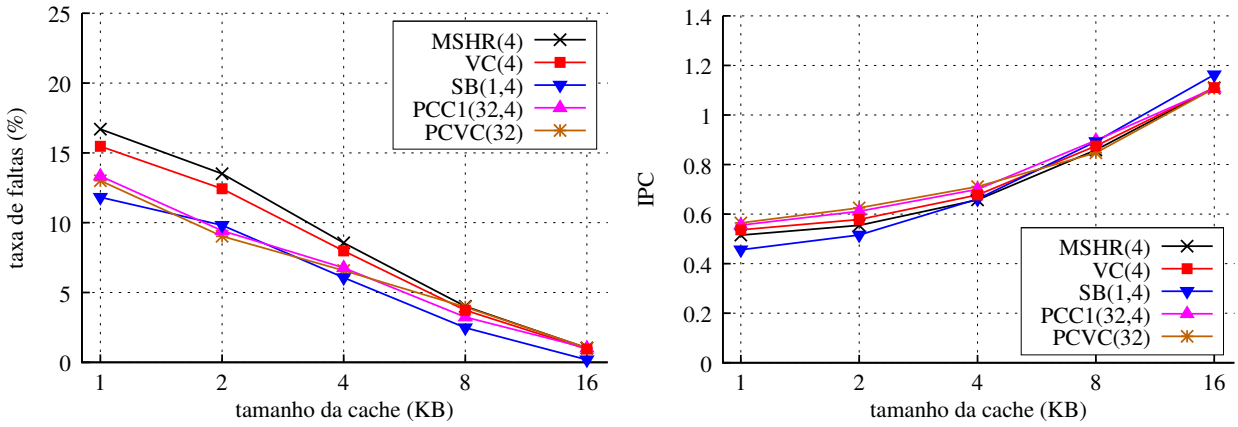

Figura 5.1: Desempenho do DRR, larg.=2, t_b=32

acréscimo das estruturas de armazenagem mais complexas.

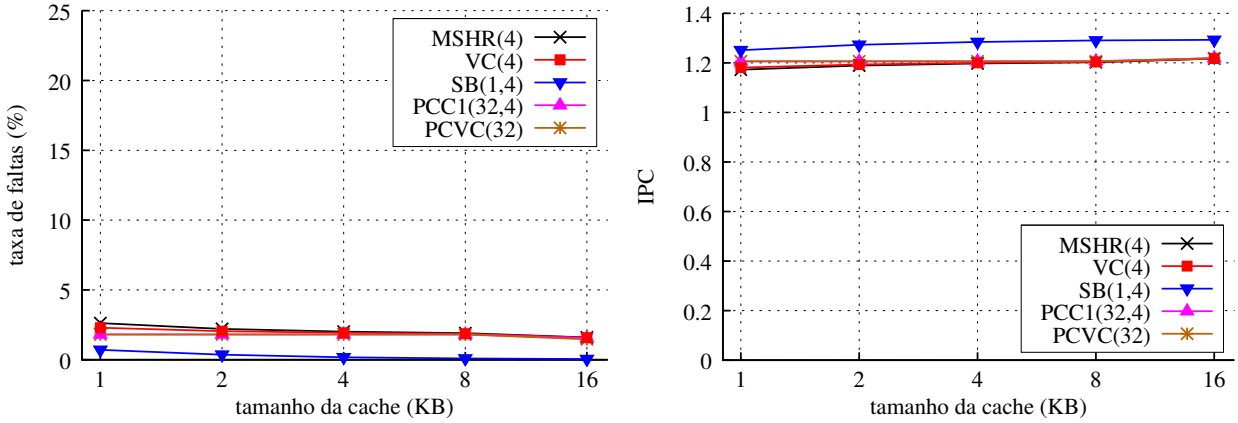

Figura 5.2: Desempenho do FRAG, larg.=2, t_b=32

O programa FRAG apresentou o resultado mais peculiar dentre os programas testados. Seu desempenho foi melhorado somente pelo uso do SB, o qual praticamente reduziu para zero a taxa de faltas e aumentou o IPC em 8%, como pode ser visto na Figura 5.2. Para todos os demais modelos de cache o desempenho é praticamente o mesmo. Isto indica que quase todas as faltas que ocorrem são do tipo compulsórias e que seu conjunto de dados é pequeno o suficiente para caber em uma cache de apenas 1K. Desta forma, somente a realização das buscas antecipadas pelo SB produz resultados positivos.

Os programas CAST e JPEG, ambos nas versões de decodificação (dec) e codificação (enc), realizam processamento sobre o conteúdo de pacotes ou informações multimídia. Seus desempenhos podem ser vistos nas Figuras 5.3, 5.4 e 5.5. Devido aos resultados do programa CAST versão ENC serem muito semelhantes aos da versão DEC, serão mostrados e descritos

apenas os resultados para esta última.

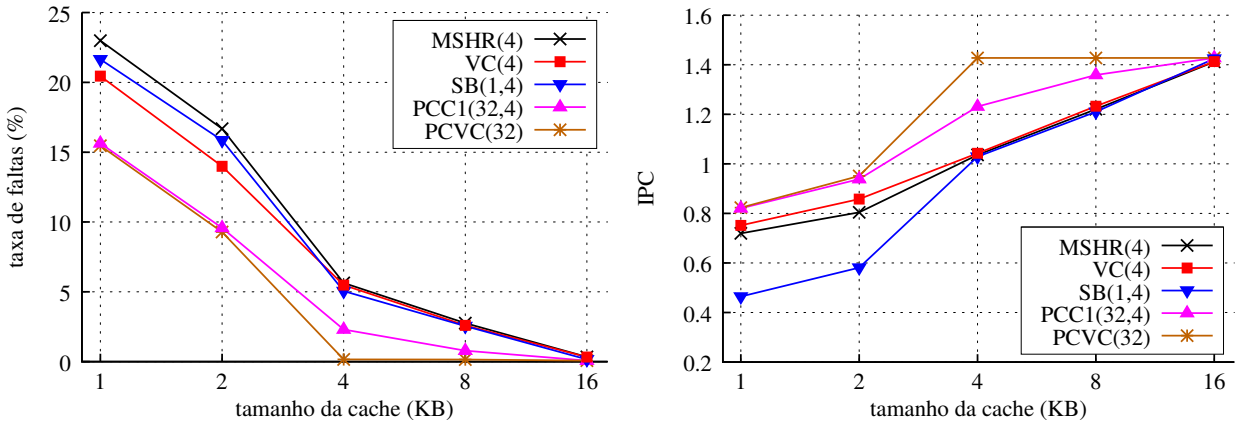

Figura 5.3: Desempenho do CAST DEC, larg.=2, t_b=32

Para os programas CAST (enc e dec), o acréscimo da VC melhora o IPC em 5% e a taxa de faltas em 13% com caches pequenas (1 e 2 KB). Com os demais tamanhos de cache a VC manteve resultados similares aos do modelo MSHR. O SB reduziu em 5% a taxa de faltas com caches pequenas mas apresentou uma queda de 35% no IPC para as mesmas caches devido à alta taxa de utilização do barramento, como descrito na Seção 4.5.2. PCC1 e PCVC apresentam ganhos de desempenho com todos os tamanhos de cache abaixo de 16 KB, com destaque para os ganhos obtidos com a cache de capacidade de 4 KB: aumentos de 19% e 38% no IPC e reduções na taxa de faltas de 59% e 97%.

A PCVC com 32 blocos praticamente reduziu para zero a taxa de faltas para o programa CAST quando operando em conjunto com uma cache de 4 KB. Isto indica que todo o conjunto de trabalho do programa cabe nesta combinação de cache e *buffer*.

Para os programas JPEG (dec e enc) mostrados nas Figuras 5.4 e 5.5, a VC apresenta pouco ganho de desempenho global; o SB reduz em cerca de 10% a taxa de faltas para todos os tamanhos de cache mas provoca uma grande queda no IPC para as caches menores que 16 KB, devido à alta taxa de utilização do barramento; a PCC1 e a PCVC obtiveram seus melhores resultados, mais de 61% e 77% de redução na taxa de faltas para uma cache de 1 KB e até 35% e 43% de ganho no IPC para o mesmo tamanho de cache.

Os ganhos na taxa de faltas e no IPC obtidos para as duas versões do programa JPEG com uma cache com capacidade próxima de 2 KB equipada com o modelo PCVC (L1 de 1

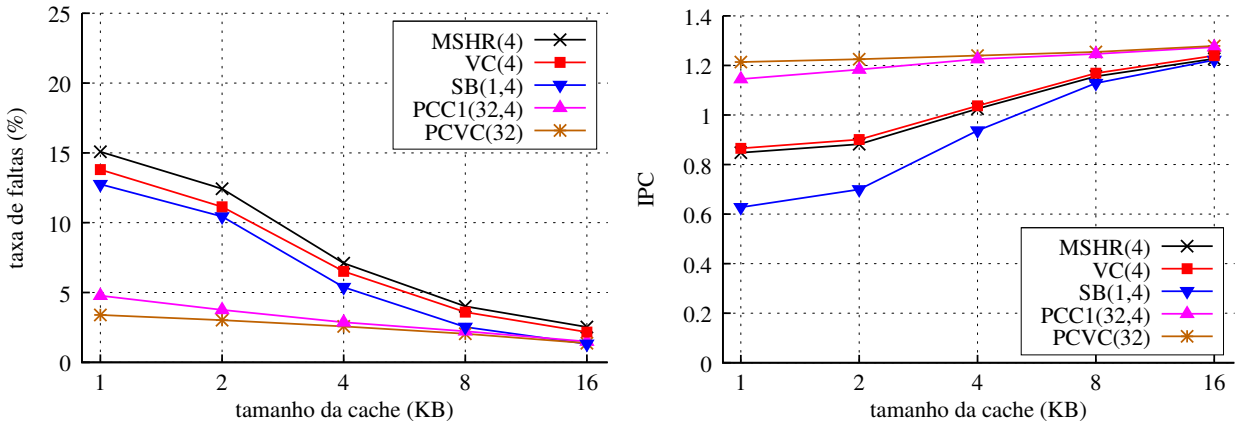

Figura 5.4: Desempenho do JPEG DEC, larg.=2, t_b=32

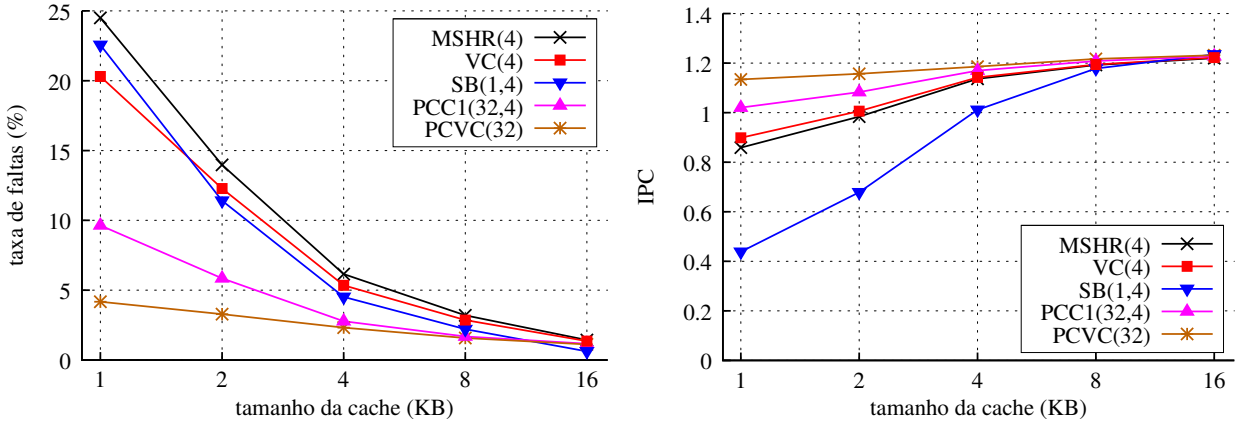

Figura 5.5: Desempenho do JPEG ENC, larg.=2, t_b=32

KB + PCVC com 32 blocos + lógica e barramentos adicionais) mostram que este modelo é muito promissor, pois os ganhos no desempenho alcançam níveis equivalentes aos de uma L1 simples de 16 KB, com oito vezes sua capacidade.

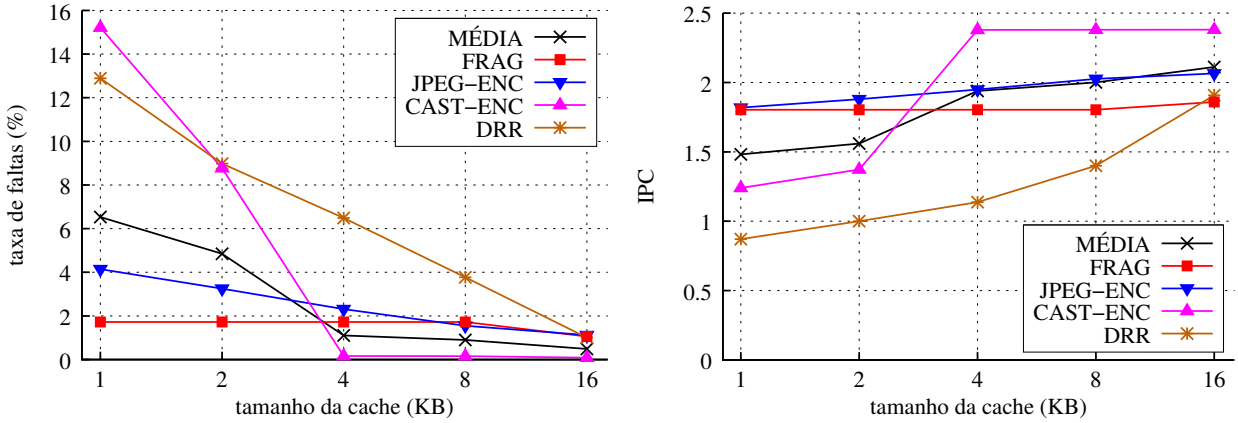

Figura 5.6: Comparativo dos programas com PCVC(32), larg.=4, t_b=32

A Figura 5.6 mostra os desempenhos individuais e a média dos programas de teste

quando executados sobre o modelo PCVC(32) em um processador de largura 4 com blocos de 32 bytes.

5.2 Padrão de Acertos no *Stream Buffer*

Com a finalidade de investigar o padrão das referências atendidas por um Stream Buffer de profundidade 8 foram adicionados ao modelo um índice de controle para cada conjunto e 9 registradores (R1-R9), além dos contadores globais de acertos e faltas. Estes índices e registradores servem apenas para contabilizar o padrão de referências, e portanto, não interferem no modelo funcional do SB.

Quando ocorre uma falta na L1 e o bloco solicitado pelo processador não está presente no SB, inicia a busca do bloco faltante e inicializa-se o índice deste conjunto com o valor zero. Em uma nova referência, caso o bloco solicitado corresponda ao que está presente na primeira posição do SB (acerto no SB), o índice do conjunto é incrementado em uma unidade, passando a apontar para o registrador R1, o qual contabiliza o acerto. Caso a referência corresponda a um bloco que está na posição β do SB ($\beta < 9$), o índice é incrementado com o valor β e o registrador correspondente contabilizará o acerto. Este processo se repete enquanto houver acertos no SB. Ainda, para índices maiores do que 8, todas as referências que resultarem em acertos no SB são contabilizadas no registrador R9.

Quando ocorre um acerto no SB, além do bloco ser entregue para a CPU e ser carregado na L1, todos os blocos subseqüentes ao recém referenciado são deslocados para a posição inicial do *buffer*, o que permite a contabilização de índices maiores do que o número de blocos no SB. Contudo, se o endereço de uma referência corresponder a um bloco além da última posição do SB, é contabilizada uma falta e o processo de busca e contagem de acertos é reinicializado. A Figura 5.7 ilustra os registradores e os blocos do SB para a seqüência de acessos descrita a seguir.

Suponha que o conjunto de blocos do SB foi recém carregado com uma nova seqüência de endereços, após uma falta na L1. O bloco na 1ª posição do SB é o bloco de endereço seguinte ao da falta. O índice é inicializado em zero. A próxima referência ocorre com uma passada igual a 2 blocos, acertando na 2ª posição do SB (bloco X na ref. 1). O índice

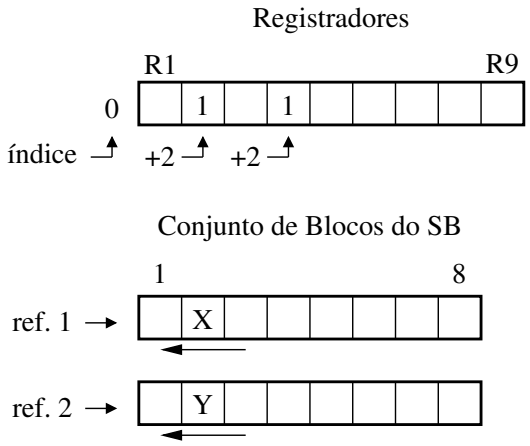

Figura 5.7: Registradores de contabilização de acertos no SB.

é incrementado de 2, e o registrador R2 acumula o acerto. Todo o SB é deslocado de 2 posições. Na próxima referência, também com passada igual a 2 blocos, ocorre novo acerto na 2ª posição do SB (bloco Y na ref. 2). O índice é incrementado novamente, o acerto será acumulado no registrador R4 e o SB é deslocado de mais 2 posições. Dessa forma, é possível registrar os padrões de referência dos programas.

A Figura 5.8 mostra o número de acertos ocorridos em cada registrador considerando blocos de largura 8, 16 e 32 bytes e SBs de 8 blocos de profundidade e associatividades 1, 2 e 4, quando executando o programa JPEG DEC com uma cache de 1 KB e processador de largura 4. Cada gráfico apresenta os resultados para uma determinada associatividade do SB. O eixo Y mostra o número de acertos em mil unidades e o eixo X mostra o tamanho dos blocos da cache e do SB, em bytes. A escala do eixo Y é mantida no mesmo valor para facilitar a comparação.

Para o SB de associatividade 1 e blocos de 8 bytes, de um total de 991 mil acertos, cerca de 286 mil ocorreram no primeiro registrador (R1), de 29 a 96 mil incidiram nos registradores entre R2 e R8, inclusive nestes, e o registrador R9 acumulou cerca de 295 mil acertos, número que indica o total de referências que ultrapassaram o oitavo bloco em acessos subseqüentes.

Para blocos de 16 bytes, tem-se que o número total de acertos caiu cerca de 25% com relação ao SB com blocos de 8 bytes mas a distribuição de acertos por registrador se manteve na mesma proporção. Com blocos de 32 bytes, a redução foi de cerca de 36% e, diferente

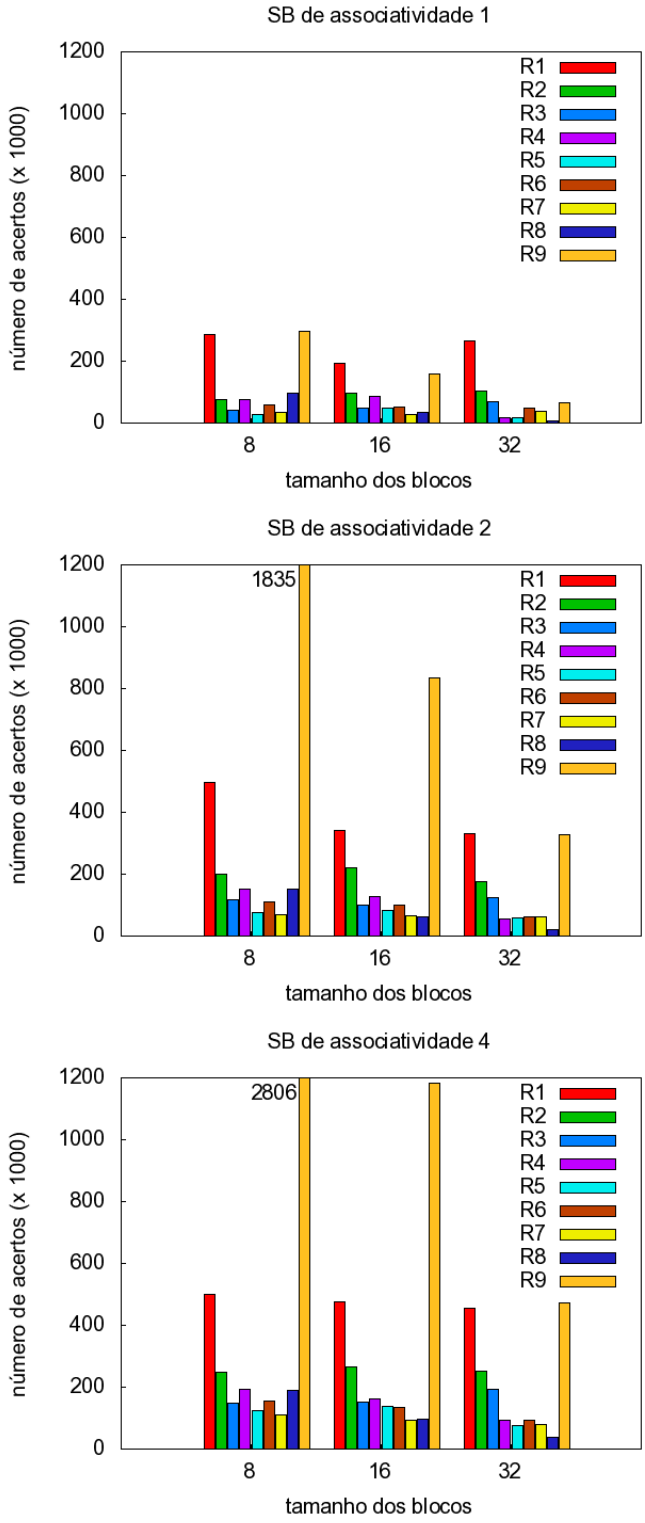

Figura 5.8: Padrão de acertos no SB com 8 blocos e assoc. 1, 2 e 4

do que ocorreu para os de 16 bytes, a proporção dos acertos nos registradores foi alterada. Para este último caso, percebe-se que o número de acertos no R1 aumentou e que os acertos acumulados no R9 diminuíram consideravelmente.

Os dois comportamentos notados —reduções (na proporção inversa ao tamanho) no número de acertos total e no número de acertos no R9— eram esperados e são intuitivos, uma vez que o aumento na largura dos blocos usa melhor a localidade espacial na cache e no SB e, consequentemente, há uma diminuição no número de referências necessárias para carga de uma mesma quantidade de dados. Quanto à largura da passada (*stride*) do acesso aos dados, não foi possível chegar a uma conclusão sobre o padrão apresentado pelo programa testado, pois, excluindo-se a grande quantidade de acertos no registrador 1, o número de referências nos demais registradores não indica uma determinada tendência em nenhum dos tamanhos de bloco.

No SB de associatividade 2 nota-se um aumento médio de 157% no número de acertos total em relação ao SB de associatividade 1, considerando-se todas as larguras de bloco, com destaque para o aumento no número de acertos no R9. Estes aumentos ocorrem devido à existência de dois caminhos de busca no SB (associatividade 2), o que permite que uma certa sequência de referências seja explorada por mais tempo no SB antes de ser substituída por uma nova sequência.

Para o SB de associatividade 4 o aumento médio no número de acertos total foi de 262% e 41% em relação aos SBs de associatividades 1 e 2 respectivamente, também considerando-se todas as larguras de bloco. Nesta configuração, as sequências de referências são ainda melhor aproveitadas do que com associatividade 2, como se pode observar comparando-se os gráficos para as associatividades 2 e 4.

Os resultados indicam que o uso da associatividade contribui muito para o ganho no desempenho do *Stream Buffer*. No entanto, deve-se lembrar que embora os SBs diminuam a taxa de faltas nas caches L1, eles degradam o desempenho global porque provocam uma grande ocupação do barramento, e portanto, seu uso é indicado para barramentos mais complexos, ou entre a cache L2 e a memória principal, onde o tráfego no barramento é menos intenso do que entre a cache L1 e a memória principal, conforme discutido na Seção 4.5.2.

5.3 PCC1 *versus* PCVC

Com base nos resultados das simulações pode-se dizer que a PCVC é um modelo mais efetivo do que a PCC1 porque ela é mais simples —possui apenas dois conjuntos de *tags* ao invés de três, é menor —somente n blocos no PCVC ao invés de PCC(n)+VC(m) blocos, e tem melhor desempenho tanto em termos de taxa de faltas quanto de IPC.

A Tabela 5.1 mostra os ganhos de desempenho da PCC1 e da PCVC quando comparadas ao modelo base (uma cache L1 com 4 MSHRs). Esta tabela permite comparar os resultados das Figuras 5.9 e 5.10, que apresentam resultados médios para os seis programas de teste da taxa de faltas e do IPC para diferentes configurações da PCC1 e da PCVC, comparadas ao modelo base e executadas com processador de largura 2 e cache com blocos de 32 bytes. Estas são as mesmas figuras utilizadas para descrever os resultados dos novos modelos no Capítulo 4. Todas as configurações de PCC1 apresentadas utilizam 4 blocos na cache de vítima.

A organização da PCC1 é dada como ⟨PCC1(*número de blocos na PCC*, *número de blocos na VC*)⟩, e para o MSHR e a PCVC o número entre parênteses indica o número de blocos.

Tamanho da Cache	1K	2K	4K	8K	16K
IPC base MSHR(4)	0,81	0,87	1,02	1,15	1,27
PCC1(4,4)/BASE	1,06	1,02	0,99	0,96	0,95
PCVC(4)/BASE	1,04	1,01	0,97	0,94	0,95
PCC1(32,4)/BASE	1,15	1,14	1,11	1,06	1,01
PCVC(32)/BASE	1,20	1,17	1,18	1,07	1,01
PCC1(32,4)/BASE∗2	—	0,96	0,86	0,86	0,85
PCVC(32)/BASE∗2	—	1,10	1,00	1,05	0,97
PCVC(32)/BASE∗4	—	—	0,95	0,89	0,95
PCVC(32)/BASE∗8	—	—	—	0,84	0,81
PCVC(32)/BASE∗16	—	—	—	—	0,78

Tabela 5.1: Ganhos no IPC da PCC1 e da PCVC

A primeira linha da tabela (MSHR(4)) mostra o IPC do modelo base para cada tamanho de cache avaliado. As próximas quatro linhas mostram o ganho no IPC para duas das configurações que fornecem ganhos com *buffers* com 4 e 32 blocos. As próximas duas linhas comparam uma dada combinação de cache+*buffer* com uma L1 simples e com o dobro da

capacidade: uma cache de 1 KB+PCVC(32) é comparada com uma L1 de 2 KB, uma de 2 KB+PCVC(32) com uma L1 de 4 KB, e assim por diante. As últimas três linhas comparam uma combinação de cache L1 com capacidade N e PCVC(32) (N KB+PCVC(32)) com caches com capacidades $4N$, $8N$ e $16N$.

Como mencionado anteriormente, as menores caches obtém ganhos relativos maiores com os *buffers*. Uma PCC1(32,4) e uma PCVC(32) aumentam o IPC de 11% a 15% e de 17% a 20%, respectivamente, para caches de 1, 2 e 4 KB comparando-se ao modelo base com 4 MSHRs.

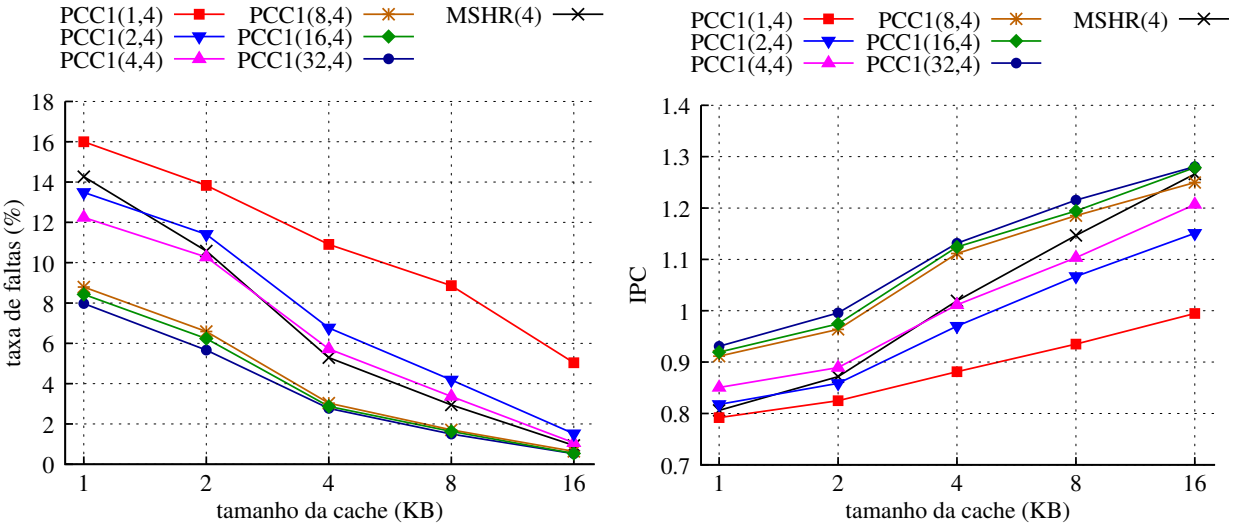

Figura 5.9: PCC1: taxa de faltas (esq.) e IPC (dir.), larg.=2, t_b=32

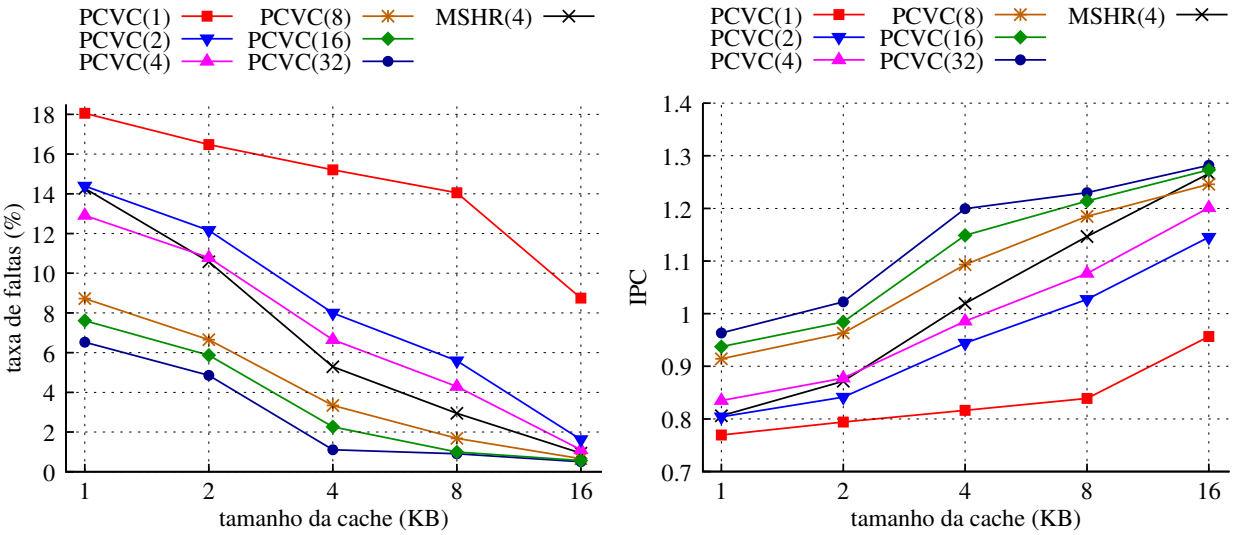

Figura 5.10: PCVC: taxa de faltas (esq.) e IPC (dir.), larg.=2, t_b=32

Nota-se nas primeiras linhas da Tabela 5.1 que os ganhos da PCC1(4,4) superam ou se

igualam aos da PCVC(4), 6% contra 4% para uma cache de 1 KB. Resultados semelhantes ocorrem para as configurações de PCC1 e PCVC com 1 ou 2 blocos, como pode ser visto nas Figuras 5.9 e 5.10. Nestas configurações, beneficiada pelo uso dos 4 blocos adicionais da VC, a PCC1 produz melhores resultados que a PCVC.

Com PCC1s e PCVCs de 8 blocos, os aumentos de desempenho destas em relação ao modelo base praticamente se igualam e, com 16 ou 32 blocos, os ganhos da PCVC superam ou se assemelham aos da PCC1, como pode ser visto nas linhas 4 e 5 da tabela e também nas figuras. Outra observação importante é que o aumento no número de blocos da PCC1, de 8 para 16 ou 32, não implica em ganho significativo de desempenho, enquanto na PCVC esta variação no número de blocos produz ganhos consideráveis.

Os resultados para a PCVC de 32 blocos mostram um ganho de 10% no IPC comparado a uma L1 simples de 2 KB (mesma capacidade da L1+PCVC) e apenas um *decréscimo* de 5% se comparado a uma L1 simples quatro vezes maior (4 KB) e com o dobro da capacidade da L1+PCVC. Para os PCC1 e PCVC de 8 blocos, os ganhos de desempenho sobre uma L1 simples de 2 KB são de 4,6% e 4,8%, respectivamente (não mostrado na tabela).

As três últimas linhas da tabela mostram que a PCVC(32) tem o melhor desempenho quando comparada a caches muito maiores. O desempenho de uma PCVC(32) com cache L1 de 1 KB corresponde a 78% do desempenho de uma cache L1 simples de 16 KB, com quatro vezes sua capacidade.

As comparações de capacidade apresentadas levam em consideração somente o espaço ocupado pelas matrizes de dados dos *buffers* e caches. A Tabela 5.2 mostra um cálculo do espaço ocupado pelas matrizes de dados do modelo PCVC e o compara com a capacidade de uma cache do modelo base com L1 duas vezes maior. A primeira linha da tabela indica que uma cache do modelo PCVC com L1 de 1 KB e com 32 blocos na PCVC —PCVC(32) $\Rightarrow$ 32 blocos de 32 bytes = 1K— ocupa 2 KB, capacidade igual à da L1 de 2 KB. Já a sexta linha indica que uma cache PCVC(16) com L1 de 4 KB é muito menor que uma L1 de 8 KB do modelo base.

Para determinar o espaço real ocupado por um sistema de memória deve-se incluir o tamanho das *tags*, dos indicadores de bloco válido e/ou sujo, bem como dos barramentos e

L1 com PCVC				L1 base
1K + PCVC(32)	$\Rightarrow$	2K	$=$	2K
2K + PCVC(32)	$\Rightarrow$	3K	$\approx$	4K
4K + PCVC(32)	$\Rightarrow$	5K	$<$	8K
1K + PCVC(16)	$\Rightarrow$	1,5K	$\approx$	2K
2K + PCVC(16)	$\Rightarrow$	2,5K	$<$	4K
4K + PCVC(16)	$\Rightarrow$	4,5K	$\ll$	8K
1K + PCVC(8)	$\Rightarrow$	1,25K	$<$	2K
2K + PCVC(8)	$\Rightarrow$	2,25K	$\ll$	4K
4K + PCVC(8)	$\Rightarrow$	4,25K	$\ll$	8K

Tabela 5.2: Capacidades do modelo PCVC comparadas com L1s do modelo base

lógica de controle das estruturas. Uma comparação mais justa depende de um projeto mais detalhado e de simulações destes projetos com ferramentas como o CACTI [27].

5.4 Composição da Taxa de Acertos Total

Os gráficos da Figura 5.11 mostram a composição da taxa de acertos para cada técnica avaliada neste estudo e permitem investigar como cada componente do modelo de cache contribui para a redução da taxa de faltas. Os dados correspondem à execução do programa JPEG DEC em um processador de largura 2 com blocos de 32 bytes e cada gráfico mostra um determinado tamanho de cache. Os dados da versão original do simulador SimpleScalar (SS_orig) foram incluídos para fins de comparação com a versão utilizada como base no estudo (MSHR(4)).

Em geral a adição de novas estruturas em uma cache L1 simples provoca uma redução no número de acertos na L1, exceto no caso do modelo SB, para o qual houve um aumento no número de acertos na L1. A redução nos acertos da L1 ocorre porque os dados que antes estavam alocados nesta, agora estão presentes em alguma outra estrutura, como por exemplo no MSHR e/ou na PCVC.

No gráfico da cache de 1 KB, ao se adicionar o modelo MSHR(4) à versão SS_orig, a taxa de acertos na L1 reduz de 77,5% para 75,8% mas esta perda é incorporada nos acertos que passam a ocorrer no MSHR (8,4%). No caso do modelo SB(1,4), comparando-se ao modelo MSHR(4), há um aumento no número de acertos na L1 e uma diminuição no MSHR. Isto ocorre porque o programa utilizado possui boa localidade espacial, e portanto, se beneficia com a busca antecipada de blocos: após um acerto na SB o bloco é carregado na L1 e as referências à endereços de palavras subseqüentes acertam todas no bloco recém carregado.

Para a cache de 16 KB há pouca alteração na taxa de acertos total, uma vez que este tamanho de cache comporta praticamente todo o conjunto de dados do programa utilizado. Em geral, nota-se apenas a migração dos acertos da L1 para as outras estruturas, exceto para o modelo SB(1,4), o qual apresenta cerca de 1,9% mais acertos do que os outros. Nas caches de maior capacidade (8 e 16 KB), o SB reduz o número de faltas compulsórias.

O modelo VC(4), como descrito anteriormente, apresenta pouco ganho na taxa de acertos (redução na taxa de faltas), cerca de 1,2% e 1,4% para caches de 1 e 2 KB, respectivamente, quando adicionado isoladamente ao modelo base. No entanto, quando incluído com uma cache de controle de poluição (modelo PCC1), produz resultados muito bons, cerca de 12,5%

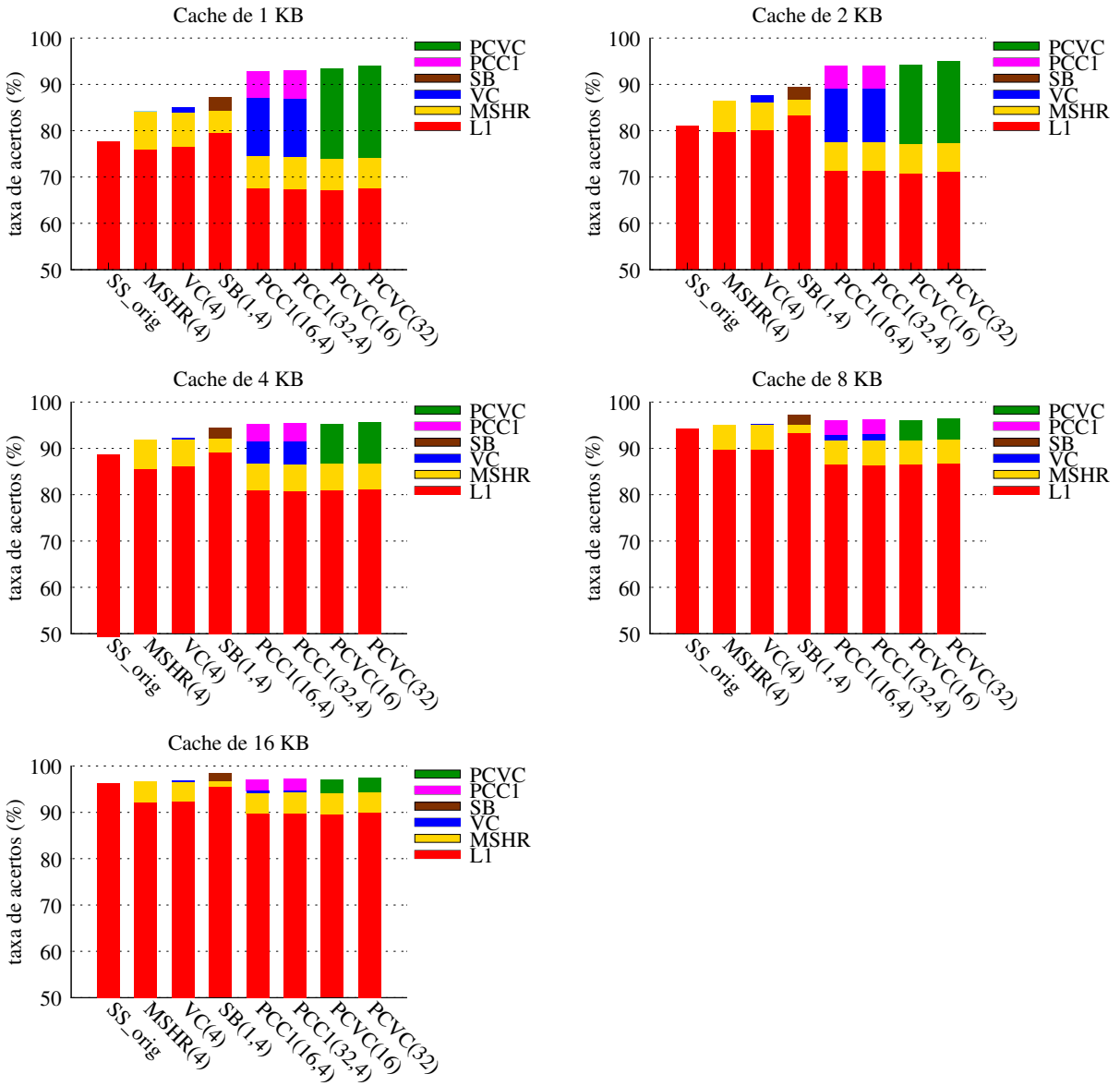

Figura 5.11: Composição da taxa de acertos para JPEG DEC, cache 1-16KB, larg.=2, t_b=32

e 11,5% para caches de 1 e 2 KB e PCC1 com 16 blocos, indicando que o controle da poluição permite um melhor aproveitamento da localidade temporal adicionada pelo uso da VC: os blocos mais freqüentemente utilizados permanecem na L1 e na VC. Estes resultados explicam o ganho de desempenho obtido com a PCVC, uma vez que esta integra as funcionalidades de controle de poluição e de cache de vítima em uma única estrutura. Com a PCVC não é necessário determinar previamente quantos blocos devem fazer parte de cada estrutura, como na PCC1, pois estes são "alocados" dinamicamente da forma que é mais eficaz para cada fase da execução, e para os diversos padrões de referência dos programas.

5.5 Tempo de Execução

A métrica definitiva da avaliação de desempenho é o tempo de execução, medido em número de ciclos. A Figura 5.12 exibe os gráficos dos números de ciclos de execução em função dos tamanhos de cache utilizados. Todos os resultados tem como base um processador de largura 2 e blocos com 32 bytes. Note que o número de ciclos é dado em milhões e que os gráficos foram gerados com escalas diferentes conforme o programa mostrado para permitir uma melhor visualização.

Efetuando-se uma análise geral dos padrões observados nos gráficos pode-se perceber o comportamento comum, e esperado, para uma memória cache L1: a redução no tempo de execução dos programas à medida que o tamanho da cache aumenta, excluindo-se o programa FRAG, que praticamente não apresenta diferença no seu desempenho ao variar-se o tamanho da cache.

Os resultados dos modelos MSHR (base) e VC se enquadram no comportamento comum, com a VC superando ou se igualando ao MSHR em todos os casos. O modelo SB apresentou os piores tempos de execução considerando-se as caches de 1, 2 e 4 KB e igualou-se ou ficou um pouco mais rápido que o modelo com MSHR para as caches de 8 e 16 KB. No caso específico do programa FRAG, o SB apresentou resultado 7% melhor do que todos os outros modelos.

A PCC1 e a PCVC, em geral, resultaram nos menores tempos de execução. Destacam-se ainda os resultados da PCVC com uma cache L1 de 4 KB nos programas CAST e JPEG, que resultam em tempos de execução muito semelhantes aos de uma cache L1 simples de 16 KB, com uma redução de mais de 2/3 na capacidade conjugada de cache e *buffers*.

Os resultados demonstram que, embora caches grandes (16 KB) possam garantir os menores tempos de execução para todos os casos, a utilização de modelos um pouco mais complexos de hierarquia de memória pode produzir desempenho similar com redução significativa no espaço ocupado.

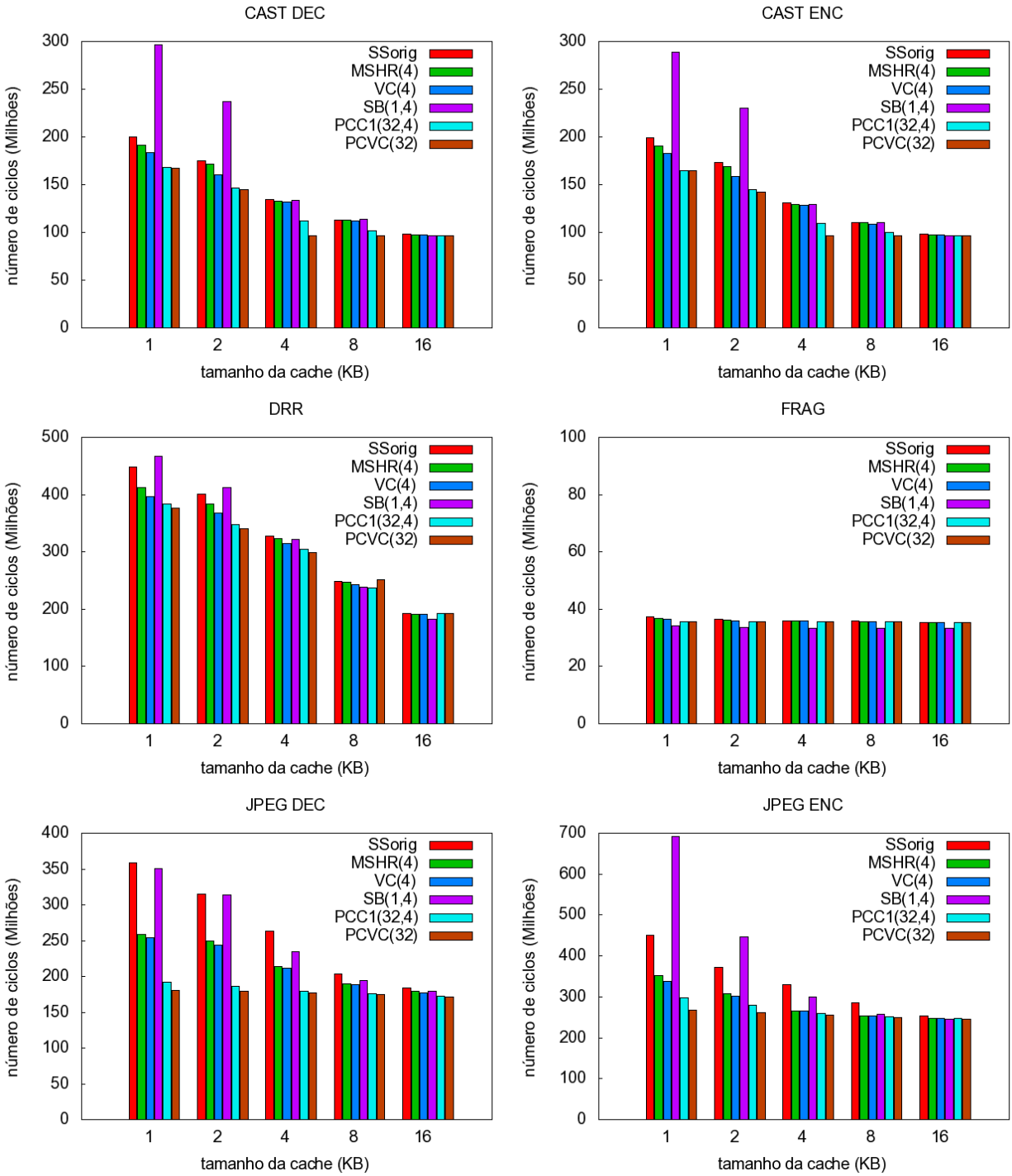

Figura 5.12: Número de ciclos total de execução dos programas, caches 1-16KB, larg.=2, t_b=32

CAPÍTULO 6

CONCLUSÃO

O projeto de caches de dados foi investigado com o intuito de obter os modelos de memória cache que produzam os melhores resultados de desempenho com os menores tamanhos, características desejadas em sistemas embarcados.

Foram investigados quatro modelos de memória cache: a Cache de Vítima (Victim Cache - VC), o Stream Buffer (SB) e duas versões da Cache de Controle de Poluição (Pollution Control Cache - PCC), nomeadas de PCC1 e PCVC, sendo esta última, a Cache de Vítima e Controle de Poluição, uma inovação proposta neste trabalho.

Em virtude do modelo de interface de memória do simulador SimpleScalar original pressupor que existe um número ilimitado de MSHRs e, portanto, o sistema de memória pode suportar infinitas faltas concorrentes (uma hipótese razoável para o projeto de processadores superescalares agressivos), foi implementada a técnica de Cache Não-Bloqueante, chamada de MSHR, para tornar o modelo de interface de memória do simulador SimpleScalar mais realístico para aplicações embarcadas e este modelo foi utilizado como base para a comparação com os demais.

Os resultados de simulação indicam que a Cache de Vítima é apropriada para aplicações que possuem um grande conjunto de dados e boa localidade temporal. Para aplicações sem boa localidade, a Cache de Vítima não apresentou bom desempenho, considerando-se os tamanhos, programas e conjuntos de dados simulados. Para uma cache L1 de 1 KB, a adição de uma VC com 16 blocos melhora a taxa de faltas em 11%, enquanto uma VC com 4 blocos melhora o IPC em 2% e 2,6% para caches de 1 e 2 KB, respectivamente, todas comparadas ao modelo base.

Para a classe de sistemas testados, o Stream Buffer teve um desempenho abaixo do esperado. Embora apresente reduções de cerca de 30% na taxa de faltas, o aumento excessivo na ocupação do barramento, ocasionado pela execução das buscas antecipadas, provoca

quedas de até 22% no IPC. Ele pode apresentar melhor desempenho em um sistema com barramento mais complexo ou se colocado entre a cache L2 e a memória principal, onde o tráfego no barramento é menos intenso do que entre a cache L1 e a memória principal.

Os testes também permitem concluir que as duas versões de Cache de Controle de Poluição testadas mostraram resultados promissores, com destaque para o novo modelo, a PCVC, a qual superou os resultados da cache mais complexa, a PCC1. Uma PCVC com 32 blocos apresentou ganhos na taxa de faltas de 54% e 79% para caches de 1 KB e 4 KB quando comparada com o modelo base utilizando os mesmos tamanhos de cache, bem como um ganho de 10% no IPC quando equipando uma cache L1 de apenas 1 KB e comparada a uma cache L1 simples de 2 KB (mesma capacidade da L1+PCVC). A PCVC obteve o melhor desempenho quando comparada a caches muito maiores. O desempenho de uma PCVC com 32 blocos e com cache L1 de 1 KB corresponde a 78% do desempenho de uma cache L1 simples de 16 KB, com quatro vezes sua capacidade.

Como resultado deste estudo, o artigo "*The Performance of Pollution Control Victim Cache for Embedded Systems*" foi aceito e apresentado no *21st Symposium on Integrated Circuits and Systems Design* (SBCCI '08)[8].

Como trabalhos futuros pretende-se estudar novos modelos de memória cache que não foram contemplados nesta pesquisa e técnicas de projeto para redução do consumo de energia dos circuitos, bem como testar o modelo PCVC com outras configurações de processadores, outros tamanhos de memórias cache e outras classes de programas de teste, como por exemplo com aplicações multimídia, pois os resultados satistatórios obtidos para o programa JPEG indicam que este modelo pode ser muito eficiente para esta classe de aplicações.

APÊNDICE A

ESTRUTURAS DE DADOS

Neste apêndice são listadas as estruturas de dados (descritas em linguagem de programação C) utilizadas para a implementação das melhorias na hierarquia de memória do simulador.

Estruturas do MSHR

```
struct mshr_t
{
    // parameters
    char *name;         // mshr name
    int nregs;          // number of registers
    int bsize;          // block size in bytes
    int nwords;         // number of words per block

    // per-mshr stats
    counter_t mshr_miss_hits;
    counter_t mshr_misses;
    counter_t mshr_full;
    counter_t mshr_hit_false;

    /* NOTE: this is a variable-size tail array, this must be the LAST field
       defined in this structure! */
    struct mshr_registers_t registers[1];
};

struct mshr_registers_t
{
    int id_reg;                 // Sequencial do registrador no MSHR

    md_addr_t addr;             // input request address
    md_addr_t tag;              // cache buffer address
    md_addr_t set;              // cache set

    int *id_fu;                 // input identification of funtional unit (one per word)
    int *send_to_cpu;           // send to CPU indicator (one per word)
    int *in_input_stack;        // in input stack indicator (one per word)
    int *partial_write;         // partial write code indicator (one per word)

    int words_processed;        // number of words of blocks processed

    int cmd;                    // command (read or write)
    int valid;                  // valid information indicator
    int obsolete;               // obsolete indicator

    tick_t ready;               // time when block will be accessible
};
```

Estruturas do Stream Buffer

```
/* sb block (or line) definition */
struct sb_blk_t
{
    struct sb_blk_t *way_next;    /* next block in the ordered way chain, used
                                     to order blocks for replacement */
    struct sb_blk_t *way_prev;    // previous block in the order way chain

    md_addr_t tag;                // data block tag value
    unsigned int status;          // block status, see SB_BLK_* defs above
    tick_t ready;                 /* time when block will be accessible, field
                                     is set when a miss fetch is initiated */
};

/* sb associative header */
struct sb_assoc_t
{
    struct sb_blk_t *way_head;  // head of way list
    struct sb_blk_t *way_tail;  // tail pf way list
    struct sb_blk_t *blks;      /* sb blocks, allocated sequentially, so this pointer
                                   can also be used for random access to sb blocks */
    tick_t lru;                 // LRU indicator

    unsigned int ind_cont;      // counter index
};

/* sb definition */
struct sb_t
{
    // parameters
    char *name;                     // sb name
    int nregs;                      // number of regs
    int bsize;                      // block size in bytes
    int assoc;                      // sb associativity
    unsigned int hit_latency;       // sb hit latency

    // miss/replacement handler
    unsigned int                                // latency of block access
        (*blk_access_fn)(enum mem_cmd cmd,      // block access command
            md_addr_t baddr,                    // program address to access
            int bsize,                          // size of the sb block
            struct cache_blk_t *blk,            // ptr to cache block struct
            tick_t now);                        // when fetch was initiated

    md_addr_t blk_mask;
    int tag_shift;
    md_addr_t tag_mask;         // use *after* shift

    tick_t bus_free;            // bus resource

    // per-sb stats
    counter_t hits;             // total number of hits
    counter_t misses;           // total number of misses
    counter_t replacements;     // total number of replacements at misses
    counter_t *cont_acertos;    // counter hits pointer

    // data blocks
```

```
    byte_t *data;               // pointer to data blocks allocation

    struct sb_assoc_t conjuntos[1];        // each entry is a header
};
```

APÊNDICE B

ROTEIRO E MODELOS DE SCRIPTS

Neste apêndice é apresentado um roteiro para preparação do ambiente de simulação da pesquisa e são listados os *scripts* de execução de experimentos, de preparação dos resultados e de criação da base de dados.

Roteiro para preparação do ambiente de simulação

Este roteiro apresenta uma lista de passos a serem seguidos para a reprodução do ambiente de execução dos experimentos e coleta de dados. Note que alguns *scripts* devem ser adequados conforme a estrutura de diretórios, os simuladores e os programas de teste utilizados. Para obtenção de todos os arquivos de configuração do ambiente de simulação desta pesquisa, entre em contato com o autor.

1. Instalar o simulador SimpleScalar (ver [3] e [21]);
2. Instalar o gerenciador de bando de dados PostgresSQL [16] e criar a base de dados para armazenamento dos resultados utilizando o *script cria_db.sql*;
3. Criar a seguinte estrutura de diretórios:

```
~/ss/
|- benchmarks          fontes e binários dos programas de teste;
   |- prog_testeA      executável do programa de teste A compilado para o SS e
   |- prog_testeB      arquivos de entrada e saída;
   ...
|- gnuplot             scripts para geração de figuras e gráficos;
|- scripts             script de extração dos dados do .out e criação do .sql;
|- simulacoes          resultados das simulações;
   |- configs          arquivos de configuração padrão dos simuladores (*.cfg);
   |- prog_testeA      scripts de execução dos experimentos do programa de teste A;
      |- resultados    resultados de simulação do programa de teste A;
         |- inserts    comandos de inserção de resultados na base agrupados;
         |- outs       backup dos resultados de simulação do SS;
         |- sqls       backup dos comandos de inserção de resultados na base;
   |- prog_testeB      scripts de execução dos experimentos do programa de teste B;
   ...
|- simuladores         executáveis das diferentes versões do simulador, como
                       por exemplo sim-outorder_base, sim-outorder_mshr_cv;
```

4. Colocar o arquivo “gera_query.awk” na pasta “*scripts*”;

5. Colocar os arquivos “.cfg” no diretório “configs”;

6. Criar um diretório para cada programa de teste a ser executado dentro da pasta “simulacoes”. Utilizar como nome da pasta o mesmo nome do executável do programa;

7. Colocar os *scripts* para execução dos simuladores dentro da pasta de cada programa de teste. Criar um *script* para cada modelo de simulador, ver exemplo de *script* listado logo abaixo;

8. Executar as simulações (Obs.: submeter os *scripts* com o comando “nohup” permite a continuidade da execução mesmo com o encerramento do terminal);

9. Ao final da execução de todas as simulações:

    9.1. Compactar os arquivos .out e salvar na pasta outs;

    9.2. Criar um arquivo único com todos os resultados das simulações para efetuar o insert na base de dados (Ex.: cat *.sql > ./inserts/nome_do_programa_versão_do_simulador.sql);

    9.3. Compactar os arquivos .sql e salvar na pasta sqls;

    9.4. Inserir os resultados das simulações na base de dados para posterior consulta.

Exemplo de *script* para execução de experimentos

```
#!/bin/bash
set -x
sfx="_$(date +%d%b%y)"
#
# Exemplo de script configurado para o programa FRAG
# Para outros programas devem ser adequados os valores dos parâmetros
#
# Simulador
modelo=mshr

SS=/home/ppginf/giancarlo/ss
sim=$SS/simuladores/sim-outorder_$modelo
p_awk=$SS/scripts/gera_query.awk

simulacoes=/home/ppginf/giancarlo/ss/simulacoes
cnf=$simulacoes/configs/config_${modelo}.cfg
res=$simulacoes/$(basename $(pwd))/resultados

if [ ! -d $res ] ; then mkdir -p $res ; fi

progr=$(basename $(pwd))
dir=$SS/benchmarks/$progr

BENCH_BIN=$dir/$progr
BENCH_OPT=$dir/words
BENCH_INP=$dir/${progr}.data
BENCH_OUT="/dev/null"
#BENCH_ARG="${BENCH_OPT} "
BENCH_ARG=" -f 576 "
# ${BENCH_INP}"

for MSHR in 1 2 4 8 16 ; do

# Exemplo para a cache de 1K
# repetir este bloco para os outros tamanhos (2K, 4K, 8K, 16K)
#
# Largura do Processador 1, 2, 4 e 8
for LARG in 1 2 4 8 ; do
    case $LARG in
        1) RUU="8"   LSQ="4"   ;;
        2) RUU="16"  LSQ="8"   ;;
        4) RUU="32"  LSQ="16"  ;;
        8) RUU="64"  LSQ="32"  ;;
    esac
# 1K
# for TEST in "16:64:1:l" "32:32:1:l" "64:16:1:l" "128:8:1:l"  ; do
#
for TEST in 01_64_1 01_32_1 01_16_1 01_08_1  ; do
#for TEST in 01_64_2 01_32_2 01_16_2 01_08_2  ; do
#for TEST in 01_64_4 01_32_4 01_16_4 01_08_4  ; do
#for TEST in 01_64_8 01_32_8 01_16_8 01_08_8  ; do
  case $TEST in
    01_64_1) c1d="16:64:1:l"  c1i="512:64:1:l"  ;;
    01_32_1) c1d="32:32:1:l"  c1i="1024:32:1:l" ;;
    01_16_1) c1d="64:16:1:l"  c1i="2048:16:1:l" ;;
    01_08_1) c1d="128:8:1:l"  c1i="4096:8:1:l"  ;;
```

```
    01_64_2) c1d="8:64:2:l"   c1i="512:64:1:l"  ;;
    01_32_2) c1d="16:32:2:l"  c1i="1024:32:1:l" ;;
    01_16_2) c1d="32:16:2:l"  c1i="2048:16:1:l" ;;
    01_08_2) c1d="64:8:2:l"   c1i="4096:8:1:l"  ;;
    01_64_4) c1d="4:64:4:l"   c1i="512:64:1:l"  ;;
    01_32_4) c1d="8:32:4:l"   c1i="1024:32:1:l" ;;
    01_16_4) c1d="16:16:4:l"  c1i="2048:16:1:l" ;;
    01_08_4) c1d="32:8:4:l"   c1i="4096:8:1:l"  ;;
    01_64_8) c1d="2:64:8:l"   c1i="512:64:1:l"  ;;
    01_32_8) c1d="4:32:8:l"   c1i="1024:32:1:l" ;;
    01_16_8) c1d="8:16:8:l"   c1i="2048:16:1:l" ;;
    01_08_8) c1d="16:8:8:l"   c1i="4096:8:1:l"  ;;
    *) echo -e "\n\n\n parametro errado $TEST \n\n\n" ; exit 1;
  esac
  echo -e "\n $TEST $c1d MSHR $MSHR LARG $LARG \n"
  $sim -config $cnf \
       -cache:dl1 dl1:$c1d \
       -cache:il1 il1:$c1i \
       -ruu:size ${RUU} \
       -lsq:size ${LSQ} \
       -cache:mshr_dl1 mshr:${MSHR} \
       -fetch:ifqsize ${LARG} \
       -decode:width ${LARG} \
       -issue:width ${LARG} \
       -commit:width ${LARG} \
       -res:ialu ${LARG} \
       -res:fpalu ${LARG} \
       -redir:sim $res/${TEST}_${modelo}_${MSHR}_LARG_${LARG}.out \
       ${BENCH_BIN} ${BENCH_ARG} < ${BENCH_INP} > ${BENCH_OUT}
  awk -f $p_awk $res/${TEST}_${modelo}_${MSHR}_LARG_${LARG}.out >
  $res/${TEST}_${modelo}_${MSHR}_LARG_${LARG}.sql
done
done

done
# MSHR
```

Script awk para preparação dos resultados para inserção na base de dados

```
#! /bin/awk
# Giancarlo C. Heck
# Data: 30/01/2008
# Arquivo: gera_query.awk

BEGIN {
  printf "INSERT INTO sim_statistics (\n"
  cp = "sim_id_caty, sim_id_lico, sim_id_bench"
  vl = " "
}

# Valor do sim_id_caty
/^sim-outorder_base:/       { vl = "1"}
/^sim-outorder_cv:/         { vl = "2"}
/^sim-outorder_sb:/         { vl = "3"}
/^sim-outorder_mshr:/       { vl = "4"}
```

```
/^sim-outorder_mshr_cv:/    { vl = "5"}
/^sim-outorder_mshr_sb:/    { vl = "6"}
/^sim-outorder_mshr_pcc1:/  { vl = "9"}
/^sim-outorder_mshr_pcc2:/  { vl = "10"}

# Valor do sim_id_lico
# Identificam a configuração básica da cache
# Exemplo: Cache de 1K, blocos de 8, 16, 32 e 64 bytes
#          e associatividade 1
/^sim:/&&/01_08_1_/{ vl = vl ", 1" }
/^sim:/&&/01_16_1_/{ vl = vl ", 2" }
/^sim:/&&/01_32_1_/{ vl = vl ", 3" }
/^sim:/&&/01_64_1_/{ vl = vl ", 4" }

# Valor do sim_id_bench
# Identificam o programa de teste
/\/cast_dec /   { vl = vl ", 1" }
/\/cast_enc /   { vl = vl ", 2" }
/\/drr /        { vl = vl ", 3" }
/\/frag /       { vl = vl ", 4" }
/\/jpeg_dec /   { vl = vl ", 5" }
/\/jpeg_enc /   { vl = vl ", 6" }

# Identificador da escalaridade do simulador
/^-fetch:ifqsize /&&/1 #/{cp = cp ", sim_id_sico" vl = vl ", 1"}
/^-fetch:ifqsize /&&/2 #/{cp = cp ", sim_id_sico" vl = vl ", 2"}
/^-fetch:ifqsize /&&/4 #/{cp = cp ", sim_id_sico" vl = vl ", 3"}
/^-fetch:ifqsize /&&/8 #/{cp = cp ", sim_id_sico" vl = vl ", 4"}

# Configurações específicas das caches
#  sim_nr_mshr
/^-cache:mshr_dl1 /&&/mshr:1 #/{cp = cp ", sim_nr_mshr" vl = vl ", 1"}
/^-cache:mshr_dl1 /&&/mshr:2 #/{cp = cp ", sim_nr_mshr" vl = vl ", 2"}
/^-cache:mshr_dl1 /&&/mshr:4 #/{cp = cp ", sim_nr_mshr" vl = vl ", 4"}
/^-cache:mshr_dl1 /&&/mshr:8 #/{cp = cp ", sim_nr_mshr" vl = vl ", 8"}
/^-cache:mshr_dl1 /&&/mshr:16 #/{cp = cp ", sim_nr_mshr" vl = vl ", 16"}
/^-cache:mshr_dl1 /&&/mshr:32 #/{cp = cp ", sim_nr_mshr" vl = vl ", 32"}

#  sim_nr_cv
/^-cache:dcv /&&/:1:l #/{cp = cp ", sim_nr_cv" vl = vl ", 1"}
/^-cache:dcv /&&/:2:l #/{cp = cp ", sim_nr_cv" vl = vl ", 2"}
/^-cache:dcv /&&/:4:l #/{cp = cp ", sim_nr_cv" vl = vl ", 4"}
/^-cache:dcv /&&/:8:l #/{cp = cp ", sim_nr_cv" vl = vl ", 8"}
/^-cache:dcv /&&/:16:l #/{cp = cp ", sim_nr_cv" vl = vl ", 16"}
/^-cache:dcv /&&/:32:l #/{cp = cp ", sim_nr_cv" vl = vl ", 32"}

#  sim_nr_sb
/^-streambuf:dsb /&&/dsb:1:/{cp = cp ", sim_nr_sb" vl = vl ", 1"}
/^-streambuf:dsb /&&/dsb:2:/{cp = cp ", sim_nr_sb" vl = vl ", 2"}
/^-streambuf:dsb /&&/dsb:4:/{cp = cp ", sim_nr_sb" vl = vl ", 4"}
/^-streambuf:dsb /&&/dsb:8:/{cp = cp ", sim_nr_sb" vl = vl ", 8"}

#  sim_assoc_sb
/^-streambuf:dsb /&&/:1 #/{cp = cp ", sim_assoc_sb" vl = vl ", 1"}
/^-streambuf:dsb /&&/:2 #/{cp = cp ", sim_assoc_sb" vl = vl ", 2"}
/^-streambuf:dsb /&&/:4 #/{cp = cp ", sim_assoc_sb" vl = vl ", 4"}
/^-streambuf:dsb /&&/:8 #/{cp = cp ", sim_assoc_sb" vl = vl ", 8"}
```

```
#  sim_nr_pcc
/^-cache:dpcc /&&/:1:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 1"}
/^-cache:dpcc /&&/:2:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 2"}
/^-cache:dpcc /&&/:4:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 4"}
/^-cache:dpcc /&&/:8:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 8"}
/^-cache:dpcc /&&/:16:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 16"}
/^-cache:dpcc /&&/:32:1 #/{cp = cp ", sim_nr_pcc" vl = vl ", 32"}

# Campos de resultados das simulações
/^sim_num_insn .*/{cp = cp ", sim_num_insn" vl = vl ", " ($2)}
/^sim_num_refs .*/{cp = cp ", sim_num_refs" vl = vl ", " ($2)}
/^sim_num_loads .*/{cp = cp ", sim_num_loads" vl = vl ", " ($2)}
/^sim_num_stores .*/{cp = cp ", sim_num_stores" vl = vl ", " ($2)}
/^sim_num_branches .*/{cp = cp ", sim_num_branches" vl = vl ", " ($2)}
/^sim_elapsed_time .*/{cp = cp ", sim_elapsed_time" vl = vl ", " ($2)}
/^sim_inst_rate .*/{cp = cp ", sim_inst_rate" vl = vl ", " ($2)}
/^sim_total_insn .*/{cp = cp ", sim_total_insn" vl = vl ", " ($2)}
/^sim_total_refs .*/{cp = cp ", sim_total_refs" vl = vl ", " ($2)}
/^sim_total_loads .*/{cp = cp ", sim_total_loads" vl = vl ", " ($2)}
/^sim_total_stores .*/{cp = cp ", sim_total_stores" vl = vl ", " ($2)}
/^sim_total_branches .*/{cp = cp ", sim_total_branches" vl = vl ", " ($2)}
/^sim_cycle .*/{cp = cp ", sim_cycle" vl = vl ", " ($2)}
/^sim_IPC .*/{cp = cp ", \"sim_IPC\"" vl = vl ", " ($2)}
/^sim_CPI .*/{cp = cp ", \"sim_CPI\"" vl = vl ", " ($2)}
/^sim_exec_BW .*/{cp = cp ", \"sim_exec_BW\"" vl = vl ", " ($2)}
/^sim_IPB .*/{cp = cp ", \"sim_IPB\"" vl = vl ", " ($2)}
/^IFQ_count .*/{cp = cp ", \"IFQ_count\"" vl = vl ", " ($2)}
/^IFQ_fcount .*/{cp = cp ", \"IFQ_fcount\"" vl = vl ", " ($2)}
/^ifq_occupancy .*/{cp = cp ", ifq_occupancy" vl = vl ", " ($2)}
/^ifq_rate .*/{cp = cp ", ifq_rate" vl = vl ", " ($2)}
/^ifq_latency .*/{ cp = cp ", ifq_latency" vl = vl ", " ($2)}
/^ifq_full .*/{ cp = cp ", ifq_full" vl = vl ", " ($2)}
/^RUU_count .*/{ cp = cp ", \"RUU_count\"" vl = vl ", " ($2)}
/^RUU_fcount .*/{ cp = cp ", \"RUU_fcount\"" vl = vl ", " ($2)}
/^ruu_occupancy .*/{ cp = cp ", ruu_occupancy" vl = vl ", " ($2)}
/^ruu_rate .*/{ cp = cp ", ruu_rate" vl = vl ", " ($2)}
/^ruu_latency .*/{ cp = cp ", ruu_latency" vl = vl ", " ($2)}
/^ruu_full .*/{ cp = cp ", ruu_full" vl = vl ", " ($2)}
/^LSQ_count .*/{ cp = cp ", \"LSQ_count\"" vl = vl ", " ($2)}
/^LSQ_fcount .*/{ cp = cp ", \"LSQ_fcount\"" vl = vl ", " ($2)}
/^lsq_occupancy .*/{ cp = cp ", lsq_occupancy" vl = vl ", " ($2)}
/^lsq_rate .*/{ cp = cp ", lsq_rate" vl = vl ", " ($2)}
/^lsq_latency .*/{ cp = cp ", lsq_latency" vl = vl ", " ($2)}
/^lsq_full .*/{ cp = cp ", lsq_full" vl = vl ", " ($2)}
/^sim_slip .*/{ cp = cp ", sim_slip" vl = vl ", " ($2)}
/^avg_sim_slip .*/{ cp = cp ", avg_sim_slip" vl = vl ", " ($2)}
/^bpred_bimod.lookups .*/{ cp = cp ", bpred_bimod_lookups" vl = vl ", " ($2)}
/^bpred_bimod.updates .*/{ cp = cp ", bpred_bimod_updates" vl = vl ", " ($2)}
/^bpred_bimod.addr_hits .*/{ cp = cp ", bpred_bimod_addr_hits" vl = vl ", " ($2)}
/^bpred_bimod.dir_hits .*/{ cp = cp ", bpred_bimod_dir_hits" vl = vl ", " ($2)}
/^bpred_bimod.misses .*/{ cp = cp ", bpred_bimod_misses" vl = vl ", " ($2)}
/^bpred_bimod.jr_hits .*/{ cp = cp ", bpred_bimod_jr_hits" vl = vl ", " ($2)}
/^bpred_bimod.jr_seen .*/{ cp = cp ", bpred_bimod_jr_seen" vl = vl ", " ($2)}
/^bpred_bimod.jr_non_ras_hits.PP .*/{ cp = cp ", \"bpred_bimod_jr_non_ras_hits_PP\""
                                      vl = vl ", " ($2)}
/^bpred_bimod.jr_non_ras_seen.PP .*/{ cp = cp ", \"bpred_bimod_jr_non_ras_seen_PP\""
                                      vl = vl ", " ($2)}
/^bpred_bimod.bpred_addr_rate .*/{ cp = cp ", bpred_bimod_bpred_addr_rate"
```

```
                                    vl = vl ", " ($2)}
/^bpred_bimod.bpred_dir_rate .*/{ cp = cp ", bpred_bimod_bpred_dir_rate"
                                   vl = vl ", " ($2)}
/^bpred_bimod.bpred_jr_rate .*/{ cp = cp ", bpred_bimod_bpred_jr_rate"
                                  vl = vl ", " ($2)}
/^bpred_bimod.bpred_jr_non_ras_rate.PP .*/{
    if ($2 ~ /<error:/){
        cp = cp ", \"bpred_bimod_bpred_jr_non_ras_rate_PP\""
        vl = vl ", 0"
    }
    else {
        cp = cp ", \"bpred_bimod_bpred_jr_non_ras_rate_PP\""
        vl = vl ", " ($2)
    }
}
/^bpred_bimod.retstack_pushes .*/{ cp = cp ", bpred_bimod_retstack_pushes"
                                    vl = vl ", " ($2)}
/^bpred_bimod.retstack_pops .*/{ cp = cp ", bpred_bimod_retstack_pops"
                                  vl = vl ", " ($2)}
/^bpred_bimod.used_ras.PP .*/{ cp = cp ", \"bpred_bimod_used_ras_PP\""
                               vl = vl ", " ($2)}
/^bpred_bimod.ras_hits.PP .*/{ cp = cp ", \"bpred_bimod_ras_hits_PP\""
                               vl = vl ", " ($2)}
/^bpred_bimod.ras_rate.PP .*/{ cp = cp ", \"bpred_bimod_ras_rate_PP\""
                               vl = vl ", " ($2)}

/^il1.accesses .*/{ cp = cp ", il1_accesses" vl = vl ", " ($2)}
/^il1.hits .*/{ cp = cp ", il1_hits" vl = vl ", " ($2)}
/^il1.misses .*/{ cp = cp ", il1_misses" vl = vl ", " ($2)}
/^il1.replacements .*/{ cp = cp ", il1_replacements" vl = vl ", " ($2)}
/^il1.writebacks .*/{ cp = cp ", il1_writebacks" vl = vl ", " ($2)}
/^il1.invalidations .*/{ cp = cp ", il1_invalidations" vl = vl ", " ($2)}
/^il1.miss_rate .*/{ cp = cp ", il1_miss_rate" vl = vl ", " ($2)}
/^il1.repl_rate .*/{ cp = cp ", il1_repl_rate" vl = vl ", " ($2)}
/^il1.wb_rate .*/{ cp = cp ", il1_wb_rate" vl = vl ", " ($2)}
/^il1.inv_rate .*/{ cp = cp ", il1_inv_rate" vl = vl ", " ($2)}

/^dl1.accesses .*/{ cp = cp ", dl1_accesses" vl = vl ", " ($2)}
/^dl1.hits .*/{ cp = cp ", dl1_hits" vl = vl ", " ($2)}
/^dl1.miss_hit_mshr .*/{ cp = cp ", dl1_miss_hit_mshr" vl = vl ", " ($2)}
/^dl1.misses .*/{ cp = cp ", dl1_misses" vl = vl ", " ($2)}
/^dl1.replacements .*/{ cp = cp ", dl1_replacements" vl = vl ", " ($2)}
/^dl1.writebacks .*/{ cp = cp ", dl1_writebacks" vl = vl ", " ($2)}
/^dl1.invalidations .*/{ cp = cp ", dl1_invalidations" vl = vl ", " ($2)}
/^dl1.miss_rate .*/{ cp = cp ", dl1_miss_rate" vl = vl ", " ($2)}
/^dl1.repl_rate .*/{ cp = cp ", dl1_repl_rate" vl = vl ", " ($2)}
/^dl1.wb_rate .*/{ cp = cp ", dl1_wb_rate" vl = vl ", " ($2)}
/^dl1.inv_rate .*/{ cp = cp ", dl1_inv_rate" vl = vl ", " ($2)}

/^dl2.accesses .*/{ cp = cp ", dl2_accesses" vl = vl ", " ($2)}
/^dl2.hits .*/{ cp = cp ", dl2_hits" vl = vl ", " ($2)}
/^dl2.miss_hit_mshr .*/{ cp = cp ", dl2_miss_hit_mshr" vl = vl ", " ($2)}
/^dl2.misses .*/{ cp = cp ", dl2_misses" vl = vl ", " ($2)}
/^dl2.replacements .*/{ cp = cp ", dl2_replacements" vl = vl ", " ($2)}
/^dl2.writebacks .*/{ cp = cp ", dl2_writebacks" vl = vl ", " ($2)}
/^dl2.invalidations .*/{ cp = cp ", dl2_invalidations" vl = vl ", " ($2)}
/^dl2.miss_rate .*/{ cp = cp ", dl2_miss_rate" vl = vl ", " ($2)}
/^dl2.repl_rate .*/{ cp = cp ", dl2_repl_rate" vl = vl ", " ($2)}
```

```
/^dl2.wb_rate .*/{ cp = cp ", dl2_wb_rate" vl = vl ", " ($2)}
/^dl2.inv_rate .*/{ cp = cp ", dl2_inv_rate" vl = vl ", " ($2)}

/^mshr.accesses .*/{ cp = cp ", mshr_accesses" vl = vl ", " ($2)}
/^mshr.miss_hits .*/{ cp = cp ", mshr_miss_hits" vl = vl ", " ($2)}
/^mshr.misses .*/{ cp = cp ", mshr_misses" vl = vl ", " ($2)}
/^mshr.full .*/{ cp = cp ", mshr_full" vl = vl ", " ($2)}
/^mshr.hits_false .*/{ cp = cp ", mshr_hits_false" vl = vl ", " ($2)}

/^dcv.accesses .*/{ cp = cp ", dcv_accesses" vl = vl ", " ($2)}
/^dcv.hits .*/{ cp = cp ", dcv_hits" vl = vl ", " ($2)}
/^dcv.misses .*/{ cp = cp ", dcv_misses" vl = vl ", " ($2)}
/^dcv.replacements .*/{ cp = cp ", dcv_replacements" vl = vl ", " ($2)}
/^dcv.writebacks .*/{ cp = cp ", dcv_writebacks" vl = vl ", " ($2)}
/^dcv.invalidations .*/{ cp = cp ", dcv_invalidations" vl = vl ", " ($2)}
/^dcv.miss_rate .*/{ cp = cp ", dcv_miss_rate" vl = vl ", " ($2)}
/^dcv.repl_rate .*/{ cp = cp ", dcv_repl_rate" vl = vl ", " ($2)}
/^dcv.wb_rate .*/{ cp = cp ", dcv_wb_rate" vl = vl ", " ($2)}
/^dcv.inv_rate .*/{ cp = cp ", dcv_inv_rate" vl = vl ", " ($2)}

/^dsb.accesses .*/{ cp = cp ", dsb_accesses" vl = vl ", " ($2)}
/^dsb.hits .*/{ cp = cp ", dsb_hits" vl = vl ", " ($2)}
/^dsb.misses .*/{ cp = cp ", dsb_misses" vl = vl ", " ($2)}
/^dsb.replacements .*/{ cp = cp ", dsb_replacements" vl = vl ", " ($2)}
/^dsb.miss_rate .*/{ cp = cp ", dsb_miss_rate" vl = vl ", " ($2)}
/^dsb.repl_rate .*/{ cp = cp ", dsb_repl_rate" vl = vl ", " ($2)}
/^dsb.bus_occupation .*/{ cp = cp ", dsb_bus_occupation" vl = vl ", " ($2)}
/^dsb.reg1_hits .*/{ cp = cp ", dsb_reg1_hits" vl = vl ", " ($2)}
/^dsb.reg2_hits .*/{ cp = cp ", dsb_reg2_hits" vl = vl ", " ($2)}
/^dsb.reg3_hits .*/{ cp = cp ", dsb_reg3_hits" vl = vl ", " ($2)}
/^dsb.reg4_hits .*/{ cp = cp ", dsb_reg4_hits" vl = vl ", " ($2)}
/^dsb.reg5_hits .*/{ cp = cp ", dsb_reg5_hits" vl = vl ", " ($2)}
/^dsb.reg6_hits .*/{ cp = cp ", dsb_reg6_hits" vl = vl ", " ($2)}
/^dsb.reg7_hits .*/{ cp = cp ", dsb_reg7_hits" vl = vl ", " ($2)}
/^dsb.reg8_hits .*/{ cp = cp ", dsb_reg8_hits" vl = vl ", " ($2)}
/^dsb.reg9_hits .*/{ cp = cp ", dsb_reg9_hits" vl = vl ", " ($2)}
/^dsb.reg10_hits .*/{ cp = cp ", dsb_reg10_hits" vl = vl ", " ($2)}
/^dsb.reg11_hits .*/{ cp = cp ", dsb_reg11_hits" vl = vl ", " ($2)}
/^dsb.reg12_hits .*/{ cp = cp ", dsb_reg12_hits" vl = vl ", " ($2)}
/^dsb.reg13_hits .*/{ cp = cp ", dsb_reg13_hits" vl = vl ", " ($2)}
/^dsb.reg14_hits .*/{ cp = cp ", dsb_reg14_hits" vl = vl ", " ($2)}
/^dsb.reg15_hits .*/{ cp = cp ", dsb_reg15_hits" vl = vl ", " ($2)}
/^dsb.reg16_hits .*/{ cp = cp ", dsb_reg16_hits" vl = vl ", " ($2)}
/^dsb.reg17_hits .*/{ cp = cp ", dsb_reg17_hits" vl = vl ", " ($2)}

/^dpcc.accesses .*/{ cp = cp ", dpcc_accesses" vl = vl ", " ($2)}
/^dpcc.hits .*/{ cp = cp ", dpcc_hits" vl = vl ", " ($2)}
/^dpcc.miss_hit_mshr .*/{ cp = cp ", dpcc_miss_hit_mshr" vl = vl ", " ($2)}
/^dpcc.misses .*/{ cp = cp ", dpcc_misses" vl = vl ", " ($2)}
/^dpcc.replacements .*/{ cp = cp ", dpcc_replacements" vl = vl ", " ($2)}
/^dpcc.writebacks .*/{ cp = cp ", dpcc_writebacks" vl = vl ", " ($2)}
/^dpcc.invalidations .*/{ cp = cp ", dpcc_invalidations" vl = vl ", " ($2)}
/^dpcc.miss_rate .*/{ cp = cp ", dpcc_miss_rate" vl = vl ", " ($2)}
/^dpcc.repl_rate .*/{ cp = cp ", dpcc_repl_rate" vl = vl ", " ($2)}
/^dpcc.wb_rate .*/{ cp = cp ", dpcc_wb_rate" vl = vl ", " ($2)}
/^dpcc.inv_rate .*/{ cp = cp ", dpcc_inv_rate" vl = vl ", " ($2)}

/^itlb.accesses .*/{ cp = cp ", itlb_accesses" vl = vl ", " ($2)}
```

```
/^itlb.hits .*/{ cp = cp ", itlb_hits" vl = vl ", " ($2)}
/^itlb.misses .*/{ cp = cp ", itlb_misses" vl = vl ", " ($2)}
/^itlb.replacements .*/{ cp = cp ", itlb_replacements" vl = vl ", " ($2)}
/^itlb.writebacks .*/{ cp = cp ", itlb_writebacks" vl = vl ", " ($2)}
/^itlb.invalidations .*/{ cp = cp ", itlb_invalidations" vl = vl ", " ($2)}
/^itlb.miss_rate .*/{ cp = cp ", itlb_miss_rate" vl = vl ", " ($2)}
/^itlb.repl_rate .*/{ cp = cp ", itlb_repl_rate" vl = vl ", " ($2)}
/^itlb.wb_rate .*/{ cp = cp ", itlb_wb_rate" vl = vl ", " ($2)}
/^itlb.inv_rate .*/{ cp = cp ", itlb_inv_rate" vl = vl ", " ($2)}
/^dtlb.accesses .*/{ cp = cp ", dtlb_accesses" vl = vl ", " ($2)}
/^dtlb.hits .*/{ cp = cp ", dtlb_hits" vl = vl ", " ($2)}
/^dtlb.misses .*/{ cp = cp ", dtlb_misses" vl = vl ", " ($2)}
/^dtlb.replacements .*/{ cp = cp ", dtlb_replacements" vl = vl ", " ($2)}
/^dtlb.writebacks .*/{ cp = cp ", dtlb_writebacks" vl = vl ", " ($2)}
/^dtlb.invalidations .*/{ cp = cp ", dtlb_invalidations" vl = vl ", " ($2)}
/^dtlb.miss_rate .*/{ cp = cp ", dtlb_miss_rate" vl = vl ", " ($2)}
/^dtlb.repl_rate .*/{ cp = cp ", dtlb_repl_rate" vl = vl ", " ($2)}
/^dtlb.wb_rate .*/{ cp = cp ", dtlb_wb_rate" vl = vl ", " ($2)}
/^dtlb.inv_rate .*/{ cp = cp ", dtlb_inv_rate" vl = vl ", " ($2)}
/^sim_invalid_addrs .*/{ cp = cp ", sim_invalid_addrs" vl = vl ", " ($2)}
/^ld_text_base .*/{ cp = cp ", ld_text_base" vl = vl ", '" ($2) "'"}
/^ld_text_size .*/{ cp = cp ", ld_text_size" vl = vl ", " ($2)}
/^ld_data_base .*/{ cp = cp ", ld_data_base" vl = vl ", '" ($2) "'"}
/^ld_data_size .*/{ cp = cp ", ld_data_size" vl = vl ", " ($2)}
/^ld_stack_base .*/{ cp = cp ", ld_stack_base" vl = vl ", '" ($2) "'"}
/^ld_stack_size .*/{ cp = cp ", ld_stack_size" vl = vl ", " ($2)}
/^ld_prog_entry .*/{ cp = cp ", ld_prog_entry" vl = vl ", '" ($2) "'"}
/^ld_environ_base .*/{ cp = cp ", ld_environ_base" vl = vl ", '" ($2) "'"}
/^ld_target_big_endian .*/{ cp = cp ", ld_target_big_endian" vl = vl ", " ($2)}
/^mem.page_count .*/{ cp = cp ", mem_page_count" vl = vl ", " ($2)}
/^mem.page_mem .*/{ cp = cp ", mem_page_mem" vl = vl ", '" ($2) "'"}
/^mem.ptab_misses .*/{ cp = cp ", mem_ptab_misses" vl = vl ", " ($2)}
/^mem.ptab_accesses .*/{ cp = cp ", mem_ptab_accesses" vl = vl ", " ($2)}
/^mem.ptab_miss_rate .*/{ cp = cp ", mem_ptab_miss_rate" vl = vl ", " ($2)}

END{
  printf cp "\n) VALUES (\n"
  printf vl "\n);\n"
}
```

Script para criação da base de dados

```
-- Arquivo: cria_db.sql

--- Database: ss
-- DROP DATABASE ss;

CREATE DATABASE ss
  WITH OWNER = postgres
       ENCODING = 'LATIN1'
       TABLESPACE = pg_default;
GRANT ALL ON DATABASE ss TO public;
GRANT ALL ON DATABASE ss TO postgres;


-- Table: benchmarks
-- DROP TABLE benchmarks;

CREATE TABLE benchmarks
(
  bench_id smallint NOT NULL, -- Identificador do programa de teste.
  bench_nome character varying(100), -- Nome do programa de teste.
  bench_tx_descricao character varying(200), -- Descrição do programa de teste.
  CONSTRAINT pk_benchmarks PRIMARY KEY (bench_id)
)
WITHOUT OIDS;
ALTER TABLE benchmarks OWNER TO postgres;
GRANT ALL ON TABLE benchmarks TO postgres;
GRANT ALL ON TABLE benchmarks TO public;


-- Table: cache_config
-- DROP TABLE cache_config;

CREATE TABLE cache_config
(
  caco_id smallint NOT NULL, -- Identificador da configuração da cache.
  caco_tx_cfg character varying(200), -- Descrição da configuração da cache.
  caco_tm_cache smallint NOT NULL, -- Tamanho da cache.
  caco_tm_bloco smallint NOT NULL, -- Tamanho do bloco.
  caco_tp_cache character varying(20) NOT NULL, -- Variação do tipo da cache.
  caco_nr_reg smallint NOT NULL DEFAULT 0, -- Número de registradores na cache especial.
  caco_nr_assoc smallint NOT NULL DEFAULT 0, -- Número da associatividade da cache.
  CONSTRAINT pk_cache_config PRIMARY KEY (caco_id, caco_tm_cache, caco_tm_bloco,
  caco_tp_cache, caco_nr_reg, caco_nr_assoc)
)
WITHOUT OIDS;
ALTER TABLE cache_config OWNER TO postgres;
GRANT ALL ON TABLE cache_config TO postgres;
GRANT ALL ON TABLE cache_config TO public;


-- Table: li_config
-- DROP TABLE li_config;

CREATE TABLE li_config
(
  lico_id smallint NOT NULL, -- Identificador da configuração da cache L1.
```

```
  lico_tx_cfg character varying(200), -- Descrição da configuração da cache L1.
  lico_tm_cache smallint NOT NULL, -- Tamanho da cache.
  lico_tm_set   smallint NOT NULL, -- Tamaho do conjunto.
  lico_tm_bloco smallint NOT NULL, -- Tamanho do bloco.
  lico_nr_assoc smallint NOT NULL DEFAULT 0, -- Número da associatividade da cache.
  CONSTRAINT pk_li_config PRIMARY KEY (lico_id, lico_tm_cache, lico_tm_set, lico_tm_bloco,
  lico_nr_assoc)
)
WITHOUT OIDS;
ALTER TABLE li_config OWNER TO postgres;
GRANT ALL ON TABLE li_config TO postgres;
GRANT ALL ON TABLE li_config TO public;


-- Table: cache_types
-- DROP TABLE cache_types;

CREATE TABLE cache_types
(
  caty_id smallint NOT NULL, -- Identificador do tipo de cache.
  caty_nm_cache character varying(100) NOT NULL, -- Nome do tipo de cache.
  caty_tx_descricao character varying(100), -- Descrição do nome da cache.
  CONSTRAINT pk_cache_types PRIMARY KEY (caty_id)
)
WITHOUT OIDS;
ALTER TABLE cache_types OWNER TO postgres;
GRANT ALL ON TABLE cache_types TO postgres;
GRANT ALL ON TABLE cache_types TO public;


-- Table: sim_config
-- DROP TABLE sim_config;

CREATE TABLE sim_config
(
  sico_id smallint NOT NULL, -- Identificador da configuração do simulador.
  sico_tx_cfg character varying(200), -- Descrição da configuração do simulador.
  sico_nr_largura smallint NOT NULL, -- Largura do processador.
  sico_nr_lsq smallint NOT NULL, -- Número de registros na LSQ.
  sico_nr_ruu smallint NOT NULL, -- Número de registros no RUU.
  sico_mem_lat character varying(10), -- Latência da memória principal.
  sico_mem_width smallint NOT NULL, -- Largura do barramento de memória.
  sico_nr_ialu smallint NOT NULL, -- Número de ALUs de inteiros.
  sico_nr_fpalu smallint NOT NULL, -- Número de ALUs de ponto flutuante.
  CONSTRAINT pk_sim_config PRIMARY KEY (sico_id)
)
WITHOUT OIDS;
ALTER TABLE sim_config OWNER TO postgres;
GRANT ALL ON TABLE sim_config TO postgres;
GRANT ALL ON TABLE sim_config TO public;


-- Table: sim_statistics
-- ALTER TABLE sim_statistics DROP CONSTRAINT pk_sim_statistics;
-- DROP TABLE sim_statistics;

CREATE TABLE sim_statistics
(
```

```
-- cp de índice
  sim_id_sico smallint NOT NULL, -- Identificador da configuração do simulador.
  sim_id_bench smallint NOT NULL, -- Identificador do programa de teste.
  sim_id_caty smallint NOT NULL, -- Identificador do tipo de cache.
  sim_id_lico smallint NOT NULL, -- Identificador da configuração da cache L1.
  sim_nr_mshr smallint NOT NULL DEFAULT 0, -- Número de registros no MSHR.
  sim_nr_cv smallint NOT NULL DEFAULT 0, -- Número de registros na CV.
  sim_nr_sb smallint NOT NULL DEFAULT 0, -- Número de registros no SB.
  sim_assoc_sb smallint NOT NULL DEFAULT 0, -- Número da associatividade do SB.
  sim_is_ce smallint NOT NULL DEFAULT 0, -- Indicador de uso da CE.
  sim_nr_pcc smallint NOT NULL DEFAULT 0, -- Número de registros na PCC.

-- cp de resultado de simulação
  sim_num_insn bigint, -- total number of instructions committed
  sim_num_refs bigint, -- total number of loads and stores committed
  sim_num_loads bigint, -- total number of loads committed
  sim_num_stores bigint, -- total number of stores committed
  sim_num_branches bigint, -- total number of branches committed
  sim_elapsed_time integer, -- total simulation time in seconds
  sim_inst_rate double precision, -- simulation speed (in insts/sec)
  sim_total_insn bigint, -- total number of instructions executed
  sim_total_refs bigint, -- total number of loads and stores executed
  sim_total_loads bigint, -- total number of loads executed
  sim_total_stores bigint, -- total number of stores executed
  sim_total_branches bigint, -- total number of branches executed
  sim_cycle bigint, -- total simulation time in cycles
  "sim_IPC" real, -- instructions per cycle
  "sim_CPI" real, -- cycles per instruction
  "sim_exec_BW" real, -- total instructions (mis-spec + committed) per cycle
  "sim_IPB" real, -- instruction per branch
  "IFQ_count" bigint, -- cumulative IFQ occupancy
  "IFQ_fcount" bigint, -- cumulative IFQ full count
  ifq_occupancy real, -- avg IFQ occupancy (insn's)
  ifq_rate real, -- avg IFQ dispatch rate (insn/cycle)
  ifq_latency real, -- avg IFQ occupant latency (cycle's)
  ifq_full real, -- fraction of time (cycle's) IFQ was full
  "RUU_count" bigint, -- cumulative RUU occupancy
  "RUU_fcount" bigint, -- cumulative RUU full count
  ruu_occupancy real, -- avg RUU occupancy (insn's)
  ruu_rate real, -- avg RUU dispatch rate (insn/cycle)
  ruu_latency real, -- avg RUU occupant latency (cycle's)
  ruu_full real, -- fraction of time (cycle's) RUU was full
  "LSQ_count" bigint, -- cumulative LSQ occupancy
  "LSQ_fcount" bigint, -- cumulative LSQ full count
  lsq_occupancy real, -- avg LSQ occupancy (insn's)
  lsq_rate real, -- avg LSQ dispatch rate (insn/cycle)
  lsq_latency real, -- avg LSQ occupant latency (cycle's)
  lsq_full real, -- fraction of time (cycle's) LSQ was full
  sim_slip bigint, -- total number of slip cycles
  avg_sim_slip real, -- the average slip between issue and retirement
  bpred_bimod_lookups bigint, -- total number of bpred lookups
  bpred_bimod_updates bigint, -- total number of updates
  bpred_bimod_addr_hits bigint, -- total number of address-predicted hits
  bpred_bimod_dir_hits bigint, -- total number of direction-predicted hits
  bpred_bimod_misses bigint, -- total number of misses
  bpred_bimod_jr_hits bigint, -- total number of address-predicted hits for JR's
  bpred_bimod_jr_seen bigint, -- total number of JR's seen
  "bpred_bimod_jr_non_ras_hits_PP" bigint, -- total number of address-predicted hits
```

```
"bpred_bimod_jr_non_ras_seen_PP" bigint, -- total number of non-RAS JR's seen
bpred_bimod_bpred_addr_rate real, -- branch address-prediction rate
bpred_bimod_bpred_dir_rate real, -- branch direction-prediction rate
bpred_bimod_bpred_jr_rate real, -- JR address-prediction rate
"bpred_bimod_bpred_jr_non_ras_rate_PP" real, -- non-RAS JR addr-pred rate
bpred_bimod_retstack_pushes bigint, -- total number of address pushed onto ret-addr stack
bpred_bimod_retstack_pops bigint, -- total number of address popped off of ret-addr stack
"bpred_bimod_used_ras_PP" bigint, -- total number of RAS predictions used
"bpred_bimod_ras_hits_PP" bigint, -- total number of RAS hits
"bpred_bimod_ras_rate_PP" real, -- RAS prediction rate

il1_accesses bigint, -- total number of accesses
il1_hits bigint, -- total number of hits
il1_misses bigint, -- total number of misses
il1_replacements bigint, -- total number of replacements
il1_writebacks bigint, -- total number of writebacks
il1_invalidations bigint, -- total number of invalidations
il1_miss_rate real, -- miss rate (i.e., misses/ref)
il1_repl_rate real, -- replacement rate (i.e., repls/ref)
il1_wb_rate real, -- writeback rate (i.e., wrbks/ref)
il1_inv_rate real, -- invalidation rate (i.e., invs/ref)

dl1_accesses bigint, -- total number of accesses
dl1_hits bigint, -- total number of hits
dl1_miss_hit_mshr bigint, -- total number of miss hit in mshrs
dl1_misses bigint, -- total number of misses
dl1_replacements bigint, -- total number of replacements
dl1_writebacks bigint, -- total number of writebacks
dl1_invalidations bigint, -- total number of invalidations
dl1_miss_rate real, -- miss rate (i.e., misses/ref)
dl1_repl_rate real, -- replacement rate (i.e., repls/ref)
dl1_wb_rate real, -- writeback rate (i.e., wrbks/ref)
dl1_inv_rate real, -- invalidation rate (i.e., invs/ref)

dl2_accesses bigint, -- total number of accesses
dl2_hits bigint, -- total number of hits
dl2_miss_hit_mshr bigint, -- total number of miss hit in mshrs
dl2_misses bigint, -- total number of misses
dl2_replacements bigint, -- total number of replacements
dl2_writebacks bigint, -- total number of writebacks
dl2_invalidations bigint, -- total number of invalidations
dl2_miss_rate real, -- miss rate (i.e., misses/ref)
dl2_repl_rate real, -- replacement rate (i.e., repls/ref)
dl2_wb_rate real, -- writeback rate (i.e., wrbks/ref)
dl2_inv_rate real, -- invalidation rate (i.e., invs/ref)

mshr_accesses bigint, -- total number of accesses
mshr_miss_hits bigint, -- total number of miss hits
mshr_misses bigint, -- total number of misses
mshr_full bigint, -- number of accesses in mshr full
mshr_hits_false bigint, -- number of hits falses

dcv_accesses bigint, -- total number of accesses
dcv_hits bigint, -- total number of hits
dcv_misses bigint, -- total number of misses
dcv_replacements bigint, -- total number of replacements
dcv_writebacks bigint, -- total number of writebacks
dcv_invalidations bigint, -- total number of invalidations
```

```
dcv_miss_rate real, -- miss rate (i.e., misses/ref)
dcv_repl_rate real, -- replacement rate (i.e., repls/ref)
dcv_wb_rate real, -- writeback rate (i.e., wrbks/ref)
dcv_inv_rate real, -- invalidation rate (i.e., invs/ref)

dsb_accesses bigint, -- total number of accesses
dsb_hits bigint, -- total number of hits
dsb_misses bigint, -- total number of misses
dsb_replacements bigint, -- total number of replacements
dsb_miss_rate real, -- miss rate (i.e., misses/ref)
dsb_repl_rate real, -- replacement rate (i.e., repls/ref)
dsb_bus_occupation bigint, -- total cicles of bus_occupation
dsb_reg1_hits bigint, -- total number of hits in registrators
dsb_reg2_hits bigint, -- total number of hits in registrators
dsb_reg3_hits bigint, -- total number of hits in registrators
dsb_reg4_hits bigint, -- total number of hits in registrators
dsb_reg5_hits bigint, -- total number of hits in registrators
dsb_reg6_hits bigint, -- total number of hits in registrators
dsb_reg7_hits bigint, -- total number of hits in registrators
dsb_reg8_hits bigint, -- total number of hits in registrators
dsb_reg9_hits bigint, -- total number of hits in registrators
dsb_reg10_hits bigint, -- total number of hits in registrators
dsb_reg11_hits bigint, -- total number of hits in registrators
dsb_reg12_hits bigint, -- total number of hits in registrators
dsb_reg13_hits bigint, -- total number of hits in registrators
dsb_reg14_hits bigint, -- total number of hits in registrators
dsb_reg15_hits bigint, -- total number of hits in registrators
dsb_reg16_hits bigint, -- total number of hits in registrators
dsb_reg17_hits bigint, -- total number of hits in registrators

dpcc_accesses bigint, -- total number of accesses
dpcc_hits bigint, -- total number of hits
dpcc_miss_hit_mshr bigint, -- total number of miss hit in mshrs
dpcc_misses bigint, -- total number of misses
dpcc_replacements bigint, -- total number of replacements
dpcc_writebacks bigint, -- total number of writebacks
dpcc_invalidations bigint, -- total number of invalidations
dpcc_miss_rate real, -- miss rate (i.e., misses/ref)
dpcc_repl_rate real, -- replacement rate (i.e., repls/ref)
dpcc_wb_rate real, -- writeback rate (i.e., wrbks/ref)
dpcc_inv_rate real, -- invalidation rate (i.e., invs/ref)

itlb_accesses bigint, -- total number of accesses
itlb_hits bigint, -- total number of hits
itlb_misses bigint, -- total number of misses
itlb_replacements bigint, -- total number of replacements
itlb_writebacks bigint, -- total number of writebacks
itlb_invalidations bigint, -- total number of invalidations
itlb_miss_rate real, -- miss rate (i.e., misses/ref)
itlb_repl_rate real, -- replacement rate (i.e., repls/ref)
itlb_wb_rate real, -- writeback rate (i.e., wrbks/ref)
itlb_inv_rate real, -- invalidation rate (i.e., invs/ref)
dtlb_accesses bigint, -- total number of accesses
dtlb_hits bigint, -- total number of hits
dtlb_misses bigint, -- total number of misses
dtlb_replacements bigint, -- total number of replacements
dtlb_writebacks bigint, -- total number of writebacks
dtlb_invalidations bigint, -- total number of invalidations
```

```
  dtlb_miss_rate real, -- miss rate (i.e., misses/ref)
  dtlb_repl_rate real, -- replacement rate (i.e., repls/ref)
  dtlb_wb_rate real, -- writeback rate (i.e., wrbks/ref)
  dtlb_inv_rate real, -- invalidation rate (i.e., invs/ref)
  sim_invalid_addrs bigint, -- total non-speculative bogus addresses seen (debug var)
  ld_text_base character varying(20), -- program text (code) segment base
  ld_text_size bigint, -- program text (code) size in bytes
  ld_data_base character varying(20), -- program initialized data segment base
  ld_data_size bigint, -- program init'ed `.data' and uninit'ed `.bss' size in bytes
  ld_stack_base character varying(20), -- program stack segment base
  ld_stack_size bigint, -- program initial stack size
  ld_prog_entry character varying(20), -- program entry point (initial PC)
  ld_environ_base character varying(20), -- program environment base address address
  ld_target_big_endian bigint, -- target executable endian-ness, non-zero if big endian
  mem_page_count bigint, -- total number of pages allocated
  mem_page_mem character varying(20), -- total size of memory pages allocated
  mem_ptab_misses bigint, -- total first level page table misses
  mem_ptab_accesses bigint, -- total page table accesses
  mem_ptab_miss_rate real -- first level page table miss rate
)
WITHOUT OIDS;

ALTER TABLE sim_statistics OWNER TO postgres;
GRANT ALL ON TABLE sim_statistics TO postgres;
GRANT ALL ON TABLE sim_statistics TO public;

ALTER TABLE sim_statistics
  ADD CONSTRAINT pk_sim_statistics PRIMARY KEY(
  sim_id_sico, sim_id_bench, sim_id_caty, sim_id_lico, sim_nr_mshr,
  sim_nr_cv, sim_nr_sb, sim_assoc_sb, sim_is_ce, sim_nr_pcc);
```

REFERÊNCIAS BIBLIOGRÁFICAS

[1] T Austin, E Larson, e D Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[2] D Burger, A Kagi, e M Hrishikesh. Memory hierarchy extensions to simplescalar 3.0. Technical Report TR99-25, Dept of Computer Sciences, The University of Texas at Austin, abril de 2000.

[3] D Burger and T M Austin. The SimpleScalar Tool Set, Version 2.0. Relatório técnico, University of Wisconsin-Madison and SimpleScalar LLC, 1997.

[4] D Tennenhouse. Proactive computing. *Communications ACM*, 43(5):43–50, 2000.

[5] K I Farkas e N P Jouppi. Complexity/performance tradeoffs with non-blocking loads. *ISCA '94: Proc 21st Intntl Symp on Computer Architecture*, páginas 211–222, 1994.

[6] Brian R Fisk e R Iris Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. *ICCD '99: Proc 1999 IEEE Intnl Conf on Computer Design*, páginas 538–545, outubro de 1999.

[7] Martin Grimheden e Martin Törngren. What is embedded systems and how should it be taught?—results from a didactic analysis. *ACM Trans on Embedded Computing Systems*, 4(3):633–651, 2005.

[8] Giancarlo C Heck e Roberto A Hexsel. The performance of pollution control victim cache for embedded systems. *SBCCI '08: 21st Symposium on Integrated Circuits and Systems Design*, páginas 1–6, setembro de 2008.

[9] G Hinton, D Sager, M Upton, D Boggs, D Carmean, A Kyker, e P Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 5, February de 2001.

[10] J L Hennessy and D A Patterson. *Arquitetura de Computadores: Uma abordagem Quantitativa*, volume 1. Campus, 3th edition, 2003.

[11] J L Hennessy and D A Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 3th edition, August de 2004.

[12] John Catsoulis. *Designing Embedded Hardware*. O'Reilly, 2th edition, 2005.

[13] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ISCA '90: Proc 17th Intnl Symp on Computer Architecture*, páginas 364–373, 1990.

[14] Norman P Jouppi. Cache write policies and performance. *ISCA '93: Proc 20th Intntl Symp on Computer Architecture*, páginas 191–201, 1993.

[15] David Kroft. Lockup-free instruction fetch/prefetch cache organization. *ISCA '81: Proc 8th Annual Symp on Computer Architecture*, páginas 81–87, 1981.

[16] PostgreSQL - Open Source Relational Database System, aug de 2007. `http://www.postgresql.org/`.

[17] R Rakvic, B Black, D Limaye, e J P Shen. Non-vital loads. *HPCA '02: Proc 8th Intntl Symp on High-Performance Computer Architecture*, páginas 165–174, 2002.

[18] J A Rivers, G S Tyson, E S Davidson, e T M Austin. On high-bandwidth data cache design for multi-issue processors. *MICRO 30: Proc 30th ACM/IEEE Intnl Symp on Microarchitecture*, páginas 46–56, 1997.

[19] Jude A Rivers e Edward S Davidson. Performance issues in integrating temporality-based caching with prefetching. *Performance Evaluation*, 27-28:189–207, 1996.

[20] Jude A Rivers e Edward S Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. *ICPP '96: Proc 1996 Intntl Conf on Parallel Processing, Vol. 1*, páginas 154–163, 1996.

[21] SimpleScalar LLC, mar de 2007. `http://www.simplescalar.com/`.

[22] Kevin Skadron e Douglas W Clark. Design issues and tradeoffs for write buffers. *HPCA '97: Proc 3rd IEEE Symp on High-Performance Computer Architecture*, páginas 144, 1997.

[23] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, 1982.

[24] Gurindar S Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans on Computers*, 39(3):349–359, 1990.

[25] M Soryani, M Sharifi, e M H Rezvani. Performance evaluation of cache memory organizations in embedded systems. *ITNG '07: 4th Intnl Conf on Information Technology 2007*, páginas 1045–1050, abril de 2007.

[26] Srikanth T Srinivasan e Alvin R Lebeck. Load latency tolerance in dynamically scheduled processors. *MICRO 31: Proc 31st ACM/IEEE Intnl Symp on Microarchitecture*, páginas 148–159, 1998.

[27] S Thoziyoor, N Muralimanohar, e N P Jouppi. Cacti 5.0: An integrated cache timing, power, and area model. Relatório técnico, HP Laboratories Palo Alto, 2007.

[28] Jim Turley. Embedded processors by the numbers. *Embedded Systems Programming*, 12(5), may de 1999. `http://www.embedded.com/1999/9905/9905turley.htm`, acessado em 15/06/2006.

[29] Stephen J Walsh e John A Board. Pollution control caching. *ICCD '95: Proc 1995 Intntl Conf on Computer Design*, páginas 300, 1995.

[30] Tilman Wolf e Mark A Franklin. CommBench - a telecommunications benchmark for network processors. *Proc IEEE Intntl Symp on Performance Analysis of Systems and Software (ISPASS)*, páginas 154–162, abril de 2000.