ROBERTO BORGES KERR JUNIOR

**Proposta e desenvolvimento de um algoritmo
de associatividade reconfigurável em
memórias cache**

Dissertação apresentada à Escola
Politécnica da Universidade de São
Paulo para obtenção do título de
Mestre em Engenharia Elétrica

São Paulo
2008

ROBERTO BORGES KERR JUNIOR

**Proposta e desenvolvimento de um algoritmo
de associatividade reconfigurável em
memórias cache**

Dissertação apresentada à Escola
Politécnica da Universidade de São
Paulo para obtenção do título de
Mestre em Engenharia Elétrica

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Edson T. Midorikawa

São Paulo
2008

FICHA CATALOGRÁFICA

Kerr Junior, Roberto Borges

Proposta e desenvolvimento de um algoritmo de associatividade reconfigurável em memórias cache / R.B. Kerr Junior. -- São Paulo, 2008.

p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Engenharia de computação 2. Arquitetura e organização de computadores 3. Arquitetura reconfigurável 4. Computação reconfigurável I. Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II. t.

DEDICATÓRIA

Dedico este trabalho a minha irmã, Elena, que foi um exemplo de vida, luta, dedicação e amor a vida à todos que a conheceram.

AGRADECIMENTOS

Agradeço aos meus pais, Roberto e Eloisa, por todo apoio, educação e conhecimento que me passaram ao longo da vida. Agradeço a minha irmã, Elise, pelos momentos de alegria e descontração.

Agradeço a minha namorada, Fulvia, por toda paciência, amor e carinho além de todo apoio e incentivo que me passou durante as fases e obstáculos vencidos deste trabalho.

Aos amigos, Paulo, Roger, Arthur, Raquel e Gustavo, que estiveram sempre presentes com sua amizade e companhia em todos os momentos. Agradeço também ao Jão e o Delane pela ajuda com a palheta de cores.

Ao orientador, Prof. Dr. Edson T. Midorikawa, pela orientação e conhecimentos transmitidos.

Agradeço a todos os amigos e familiares que estiveram presentes neste momento, ajudando direta ou indiretamente a realização de um sonho.

*"There is no reason
anyone would want a
computer in their home."
(Ken Olson)*

RESUMO

A evolução constante dos processadores está aumentando cada vez o *overhead* dos acessos à memória. Tentando evitar este problema, os desenvolvedores de processadores utilizam diversas técnicas, entre elas, o emprego de memórias cache na hierarquia de memórias dos computadores. As memórias cache, por outro lado, não conseguem suprir totalmente as suas necessidades, sendo interessante alguma técnica que tornasse possível aproveitar melhor a memória cache. Para resolver este problema, autores propõem a utilização de técnicas de computação reconfigurável. Este trabalho analisa um trabalho na área de reconfiguração na associatividade de memórias cache, e propõe melhorias nele para uma melhor utilização de seus recursos, apresentando resultados práticos de simulações realizadas com diversas organizações de cache.

Palavras-chave: Engenharia de computação. Arquitetura e organização de computadores. Arquitetura reconfigurável. Computação reconfigurável.

ABSTRACT

With the constant evolution of processor's architecture, it's getting even bigger the overhead generated with memory access. Trying to avoid this problem, some processor's developers are using several techniques to improve the performance, as the use of cache memories. By the otherside, cache memories cannot supply all their needs, that's why it's important some new technique that could use better the cache memory. Working on this problem, some authors are using reconfigurable computing to improve the cache memory's performance. This work analyses the reconfiguration of the cache memory associativity algorithm, and propose some improvements on this algorithm to better use it's resources, showing some practical results from simulations with several cache organizations.

Keywords: Computer engineering. Computer architecture and organization. Reconfigurable architecture. Reconfigurable computing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo de uma linha de memória cache	8
Figura 2 – Exemplo de um controlador de memória cache	9
Figura 3 – Mapeamento direto.....	11
Figura 4 – Mapeamento totalmente associativo.....	12
Figura 5 – Mapeamento Associativo por conjuntos.....	12
Figura 6 – Arquitetura Von-Neumann vs. Arquitetura Harvard.....	15
Figura 7 – Hierarquia de memórias.....	17
Figura 8 – Arquitetura de cache reconfigurável.....	30
Figura 9 – Exemplo de acessos não uniformes à memória	31
Figura 10 – Exemplo de doação/recepção de linhas entre conjuntos.....	32
Figura 11 – Conjunto de linhas da memória cache proposta	33
Figura 12 – Rótulos para classificação.....	34
Figura 13 – Formato dos arquivos de log.....	42
Figura 14 – Formato dos arquivos de log.....	44
Figura 15 – Modelo self-scheduling	45
Figura 16 – Desempenho de diferentes caches	48
Figura 17 – Análise da influência da inversão no padrão doador-receptor	49
Figura 18 – Influência da alteração no padrão doador-receptor nas simulações realizadas.....	50
Figura 19 – Influência da prioridade diferenciada na doação e recepção de linhas nas simulações realizadas	52
Figura 20 – Configuração 4-8 way.....	54
Figura 21 – Cache de instruções e de dados.....	57
Figura 22 – Resultados com cache de 32KB não unificadas com gzip.....	59
Figura 23 – Resultados com cache de 64KB não unificadas com gzip.....	60
Figura 24 – Resultados com cache de 32KB não unificadas com gcc.....	61
Figura 25 – Resultados com cache de 64KB não unificadas com gcc.....	62
Figura 26 – Resultados com cache de 32KB não unificadas com bzip2.....	64
Figura 27 – Resultados com cache de 64KB não unificadas com bzip2.....	65
Figura 28 – Outros resultados com cache de 32KB não unificadas	66
Figura 29 – Outros resultados com cache de 64KB não unificadas	68
Figura 30 – Comparação entre L1 e L2 de mesmas características.....	71
Figura 31 – Resultados com cache em L2 utilizando gzip.....	73
Figura 32 – Resultados com cache em L2 utilizando gcc.....	74
Figura 33 – Resultados com cache em L2 utilizando bzip2.....	76
Figura 34 – Outros resultados com cache em L2.....	77

LISTA DE TABELAS

Tabela 1 – Descrição de rótulos	26
Tabela 2 – Exemplo de classificação	27
Tabela 3 – Exemplo de rotulação	35
Tabela 4 – Classificação	35
Tabela 5 – Benchmarks utilizados nas simulações	39

SUMÁRIO

1 INTRODUÇÃO	1
1.1 Motivação	2
1.2 Objetivo e Contribuições Esperadas	3
1.3 Metodologia	3
1.4 Organização do Trabalho	5
2 ARQUITETURAS ANALISADAS	6
2.1 Memórias Cache	6
2.1.1 Princípio da Localidade	7
2.1.2 Analogia da Biblioteca	7
2.1.3 Arquitetura do controlador	8
2.1.4 Associatividade	10
2.1.4.1 Mapeamento direto	10
2.1.4.2 Mapeamento totalmente associativo	11
2.1.4.3 Mapeamento associativo por conjunto	12
2.1.5 Políticas da memória cache	13
2.1.5.1 Algoritmos de substituição	13
2.1.5.2 Estratégias de escrita	15
2.1.6 Memórias cache unificadas e não-unificadas	15
2.1.7 Memórias cache multi-níveis	16
2.2 Computação Reconfigurável	18
2.2.1 Conceituação	19
2.2.2 Dispositivos Reconfiguráveis	20
2.2.3 Processadores e a Reconfiguração	21
2.3 Cache Reconfigurável	23
2.3.1 Reconfigurabilidade em memórias caches	23
2.3.2 Reconfiguração na Associatividade	25
2.3.2.1 Algoritmo Carvalho e Martins	25
3 PROPOSTA DE RECONFIGURAÇÃO DA ASSOCIATIVIDADE	29
3.1 Deficiências nas propostas atuais	29
3.2 Proposta do novo algoritmo	29

4 SIMULADOR DESENVOLVIDO	38
4.1 Estudo de simuladores e traces.....	38
4.2 Simulador Bobsim.....	40
4.2.1 Bobsim versão 1.0.....	40
4.2.2 Bobsim versão 2.0.....	42
4.3 Ambiente computacional.....	46
5 MEMÓRIAS CACHE UNIFICADAS.....	47
5.1 Simulações Realizadas.....	47
5.1.1 Caso 1: Padrão receptor-doador	48
5.1.2 Caso 2: Prioridade diferenciada na recepção e doação.....	51
5.1.3 Caso 3: Configuração 4-8 way	53
5.2 Conclusões dos resultados obtidos	55
6 MEMÓRIAS CACHE NÃO UNIFICADAS.....	56
6.1 Simulações realizadas	56
6.1.1 Resultados obtidos.....	56
6.1.1.1 Simulações com <i>traces</i> do gzip	58
6.1.1.2 Simulações com <i>traces</i> do gcc	61
6.1.1.3 Simulações com <i>traces</i> do bzip2	63
6.1.1.4 Outras simulações	65
6.2 Conclusões dos resultados obtidos	68
7 MEMÓRIAS MULTI-NÍVEIS.....	70
7.1 Simulações realizadas	70
7.1.1 Resultados obtidos.....	71
7.1.1.1 Simulações com <i>traces</i> do gzip	72
7.1.1.2 Simulações com <i>traces</i> do gcc	73
7.1.1.3 Simulações com <i>traces</i> do bzip2	75
7.1.1.4 Outras simulações	76
7.2 Conclusões dos resultados obtidos	78
8 CONCLUSÕES FINAIS E TRABALHOS FUTUROS	80
8.1 Conclusões finais.....	80
8.2 Trabalhos Futuros.....	83
REFERÊNCIAS.....	84

1 INTRODUÇÃO

O aumento da capacidade computacional está entre uma das maiores preocupações dos desenvolvedores de processadores. Várias inovações têm sido introduzidas para obter melhoras de desempenho, seja com novas técnicas como os processadores superescalares e VLIW e as arquiteturas *multicore*, seja com mudanças mais diretas como o aumento no *clock* [1]. Mas essas constantes evoluções acabam causando efeitos indesejados em sistemas computacionais, como é o exemplo da alta latência no acesso à memória criada pelo aumento na frequência de funcionamento do processador, enquanto a frequência de funcionamento das memórias não seguiu essa evolução tornando o acesso à memória um dos maiores gargalos dos sistemas [2].

Tentando resolver esta deficiência, no final da década de 60, foi proposta a utilização de uma memória de baixa latência junto ao processador, conhecida como memória cache [3]. Pelo seu custo elevado, a memória cache possui um tamanho bem limitado, mas seu funcionamento tem sua base na heurística em que o processador gasta cerca de 90% do tempo de processamento em apenas 10% do código [1]. Desta forma, caches com tamanhos pequenos, tais como 32KB e 64KB, têm ganhos no desempenho muito convincentes.

Por outro lado, buscando o aumento no desempenho na realização de operações específicas como a execução de algoritmos de criptografia e a compactação de vídeos, a utilização de *hardwares* de uso específico, como as ASICs, ganhou espaço, principalmente por causa do seu alto desempenho.

As ASICs se mostraram bastante eficientes para a realização de tarefas específicas, mas a impossibilidade da utilização destes dispositivos em outras tarefas, por ser um *hardware* produzido especialmente para a realização de uma operação definida em projeto, motivou a utilização de sistemas baseados em dispositivos reconfiguráveis, como as FPGAs. Com a principal característica de poder ser reconfigurado a diversas tarefas alterando sua configuração, os dispositivos reconfiguráveis têm ganhado cada vez mais espaço entre os projetistas de *hardware*. Projetos diversos vêm utilizando FPGAs para a realização das mais inusitadas tarefas, e em alguns casos, estão sendo utilizados como co-processadores que operam em paralelo aos processadores de *hardware* fixo, no

intuito de realizar operações específicas como a execução de algoritmos de criptografia entre outros.

O crescimento da utilização da computação reconfigurável [4] abriu possibilidades da utilização dela nos mais diversos campos da computação. Trabalhos vêm sendo realizados para a implementação de reconfigurabilidade em memórias cache, visando os mais diversos objetivos, como a diminuição da potência dissipada ou até a diminuição do gargalo criado entre processador e memória.

1.1 Motivação

A utilização de memórias cache nos processadores atuais se tornou indispensável devido a problemas gerados com a evolução dos *hardwares*, principalmente em relação à defasagem gerada pela evolução constante do fluxo de informações que um processador consegue gerar atualmente aliada a alta latência dos níveis hierárquicos mais baixos de memória. Porém as memórias caches comuns utilizadas atualmente têm uma grande deficiência por causa da subutilização dos seus conjuntos de linhas. Esta deficiência está diretamente relacionada com a não uniformidade dos acessos à memória.

Na tentativa de diminuir ainda mais o gargalo entre processador e memória, uma das técnicas que se sobressaem é a utilização dos conceitos de computação reconfigurável nos níveis hierárquicos de memória com o intuito de uma melhor utilização.

Diversos trabalhos estão sendo propostos nesta área visando a reconfiguração das memórias cache durante a operação. Conhecida por reconfiguração dinâmica parcial¹ da memória cache, pois realiza uma reestruturação na configuração na memória cache durante o seu funcionamento e de forma parcial, ou seja, apenas em alguns módulos da memória cache sobre a reconfiguração.

Nossa principal motivação para o desenvolvimento de nosso trabalho está na proposta de um algoritmo de memória reconfigurável que possa melhorar o desempenho da memória cache com a adaptação desta memória a diversas cargas de trabalho, visando uma melhor utilização de seus recursos.

¹ Apesar de não haver trabalhos relacionados utilizando a reconfiguração dinâmica total das memórias cache, esta técnica também poderia ser utilizada.

1.2 Objetivo e Contribuições Esperadas

Este trabalho tem como objetivo a proposta e análise de um algoritmo de reconfiguração da associatividade de memórias cache de acordo com a carga de trabalho. Para isso, iremos analisar propostas de arquiteturas de cache reconfiguráveis buscando melhorias sobre as deficiências apresentadas por elas juntamente com a proposta de sistemas de adaptatividade não utilizados anteriormente. Nossas análises sobre o desempenho da arquitetura baseada em nossa proposta de algoritmo serão realizadas através de simulações. Com isso esperamos o desenvolvimento de um algoritmo reconfigurável mais aperfeiçoado para realizar a sua adaptatividade a diversas cargas de trabalho.

Este trabalho contribui nos seguintes aspectos:

- Revisão bibliográfica sobre memórias cache reconfiguráveis;
- Análise das arquiteturas reconfiguráveis;
- Discussão detalhada sobre um algoritmo de reconfiguração da associatividade de memórias cache da literatura;
- Proposta e avaliação de desempenho de um novo algoritmo de reconfiguração da associatividade em diversas configurações de memórias caches.

Com o cumprimento dos objetivos deste trabalho, a utilização de reconfiguração da associatividade das memórias cache permitirá maximizar o seu desempenho independente da carga de trabalho aplicada, cumprindo com nossas metas de trabalho.

1.3 Metodologia

Nosso método de trabalho foi dividido em: estudo bibliográfico, análise de dispositivos e simuladores, e proposta de algoritmo e desenvolvimento de simuladores.

Nosso primeiro ano foi dedicado ao cumprimento dos créditos das disciplinas e ao estudo bibliográfico, onde buscamos inicialmente a definição de conceitos como computação reconfigurável, analisando trabalhos realizados com definições e abordagens mais concretas sobre o assunto [5][6], podendo assim fixar um conceito próprio sobre a computação reconfigurável.

Após este estudo, partiu-se para uma análise e pesquisa de diversos

trabalhos que estivessem utilizando reconfiguração em processadores e também a reconfiguração em memórias cache. Após este estudo inicial de projetos utilizando reconfiguração em memórias cache, partimos para a dedicação de nossos estudos bibliográficos a análise de trabalhos utilizando a reconfiguração dinâmica do algoritmo de substituição de linhas das memórias cache, onde estudos foram realizados tanto com algoritmos de substituição de linhas em memórias cache de processadores, como os algoritmos de substituição de páginas em sistemas operacionais [7][8]. Este estudo rendeu o trabalho apresentado durante uma das disciplinas cursadas², fazendo a análise de diversos algoritmos.

Após a conclusão dos créditos das disciplinas, estudos foram realizados em busca de um conhecimento melhor sobre os dispositivos reconfiguráveis, tanto quanto ambientes de simulação de processadores. Inicialmente, testes e simulações foram realizados utilizando o kit de desenvolvimento da Digilent [9] com FPGA Spartan 3 da Xilinx. A implementação de sistemas e a modificação de *hardware* utilizando a linguagem de descrição de *hardware* VHDL [10] foi o principal foco desta parte dos estudos, onde pode se ter uma melhor noção da estrutura interna além da possibilidade de uma melhor utilização do dispositivo reconfigurável em questão.

Em paralelo a estas simulações, foram realizados estudos aprofundados sobre a estrutura de simuladores de processadores conhecidos. Inicialmente, uma análise sobre a estrutura do *software* do SimpleScalar [11] nos rendeu modificações significativas nas simulações, com a implementação de algoritmos de substituição de páginas para a realização das simulações. Após os estudos realizados com este *software*, foram analisados processadores escritos em VHDL. Inicialmente estudamos uma implementação em VHDL do processador proposto por Hennessy e Patterson, o DLX [1], e posteriormente, o processador Leon 2 e sua versão subsequente, o Leon 3 [12]. Simulações utilizando estas implementações em VHDL foram realizadas tanto no simulador GHDL [13] quanto no ModelSim [14] utilizando cargas de trabalho reais como uma versão reduzida do Linux [15] pré-compilada para o Leon 3 que pode ser encontrada disponível para *download* no site deste projeto.

Por fim, um estudo detalhado sobre a utilização de *traces* para a simulação de acessos à memória foi realizado. Inicialmente, o código fonte do simulador de cache

²

PCS 5720 - Sistemas Operacionais

Dinero IV [16] foi analisado. Após este estudo, desenvolvemos um simulador de memórias cache comuns com diversos algoritmos de substituição de páginas, que usasse como carga de trabalho *traces* obtidos pelo BYU Trace Distribution Center [17].

Após a definição do simulador e após seu desenvolvimento, análises sobre as deficiências de propostas de algoritmos de reconfiguração de memórias caches foram realizadas, e a partir destas análises, realizamos uma nova proposta de algoritmo de reconfiguração de associatividade de memórias cache. A implementação desta proposta foi realizada utilizando o simulador desenvolvido por nós.

A última fase de nosso trabalho foi a simulação de nossa proposta de arquitetura com diversos *traces* dos *benchmarks* do Spec2000 obtidos no BYU Distribution Center. Com os resultados destas simulações, foi publicado um artigo no o VIII Workshop em Sistemas Computacionais de Alto Desempenho, WSCAD 2007 [18].

1.4 Organização do Trabalho

No próximo capítulo será apresentada uma introdução sobre a utilização das memórias cache e seu funcionamento, seguido de uma explicação detalhada sobre computação reconfigurável e seus conceitos junto com alguns exemplos de dispositivos configuráveis, e por fim será feita uma análise sobre a situação atual das memórias cache reconfiguráveis e seus tipos de reconfigurabilidade. No capítulo 3, iremos mostrar algumas deficiências sobre um modelo de memória cache reconfigurável juntamente com a proposta de uma memória cache reconfigurável que supre essas deficiências. No capítulo 4 é descrito o ambiente de simulação juntamente com o simulador desenvolvido para as análises. No capítulo 5 serão apresentados alguns dos resultados obtidos neste ambiente com a análise de três casos especiais. No capítulo 6 serão realizadas algumas análises comparativas de resultados obtidos com o simulador desenvolvido para memórias cache não unificadas. O capítulo 7 será destinado a análises comparativas dos resultados obtidos para os diversos algoritmos de associatividade em memórias cache multi-níveis, e concluímos no capítulo 8 com a apresentação das conclusões finais de nosso trabalho, juntamente com as propostas de trabalhos futuros.

2 ARQUITETURAS ANALISADAS

Neste capítulo iremos fazer inicialmente uma análise sobre as memórias caches explicando a sua utilidade nos processadores e o funcionamento de seu controlador juntamente com as possíveis políticas a serem implementadas em sua arquitetura. Em seguida, uma análise detalhada sobre os conceitos da computação reconfigurável e os dispositivos reconfiguráveis, entre eles, os processadores reconfiguráveis. Além disso, será introduzido o conceito de caches reconfiguráveis com os possíveis campos de se realizar reconfiguração, dentre eles, a associatividade. No final deste capítulo será apresentado um algoritmo que realiza reconfiguração em associatividade.

2.1 Memórias Cache

Memória cache é uma técnica inicialmente implementada em sistemas operacionais, que visava estender a capacidade de memória principal do computador, usando espaço em disco para manter as informações menos acessadas, enquanto a memória possuía os dados mais recentes para acesso mais rápido, seria uma forma de duplicar o tamanho da memória usando uma tecnologia mais barata, porém lenta.

Com a evolução constante dos processadores, e a tecnologia de acesso a dados da memória ficando cada vez mais lenta em relação à velocidade dos processadores, tanto pela tecnologia utilizada nos dispositivos de memória que não pode seguir a utilizada em processadores devido ao seu custo, e pelo ao atraso no acesso devido à inclusão de *buffers* que tem um atraso de propagação fixo, não melhorando com os avanços da microeletrônica [3], aumentou-se cada vez mais a diferença de velocidade de operação entre o processador e a memória, tornando esta o fator limitante no aumento do fluxo de informações.

Por causa deste problema, resolveu-se adicionar uma memória mais próxima ao processador, utilizando uma tecnologia mais rápida, para guardar as informações mais acessadas, diminuindo a necessidade de acesso à memória principal, o que aumentaria o desempenho total do sistema.

Em 1968, o IBM System/360 Model 85 [19] implementa pela primeira vez a utilização de uma memória cache de processadores [20], fazendo dados de alta

prioridade estarem disponíveis até 12 vezes mais rápidos do que anteriormente.

2.1.1 Princípio da Localidade

O princípio de localidade tem uma grande importância na utilização de memória cache, pois toda a base da utilização deste tipo de memória parte deste princípio. Por volta dos anos 60, pesquisadores da IBM descobriram que quase todo o código dos programas era extremamente repetitivo [3], os programas costumam gastar aproximadamente 90% do tempo de execução em apenas 10% do código [1]. Dois conceitos de localidades foram definidos, a localidade espacial e a localidade temporal.

A localidade espacial está ligada a idéia que os programas são, antes de tudo, um conjunto de códigos sequenciais, ou seja, se um endereço de memória foi acessado, muito provavelmente o endereço seguinte também será acessado, como pode ser observado no código a seguir:

1000	MOVEQ	#9,D0
1004	CLR.L	D1
1008	MOVEQ	#5,D2
100C	ADD.L	D2,D1
1010	ADD.L	D4,D7
1018	SUBI.L	D5,D6
1014	DBF	D0,\$100C
101C	NOP	
1020	NOP	
1024	JMP	\$1024

O princípio da localidade temporal utiliza o conceito de que é muito mais provável que um trecho do código recém executado deva ser executado em breve outra vez do que um trecho executado há muito tempo atrás. Este conceito pode ser visto no código demonstrado, onde o trecho do *loop* será executado por 100 vezes até que o programa siga em frente. Estes conceitos foram demonstrados em um pequeno trecho de um código demonstrativo, porém eles são totalmente válidos em programas maiores, iguais ao utilizados hoje em dia.

2.1.2 Analogia da Biblioteca

Para entender melhor a utilização das memórias caches em um sistema microprocessador pode ser utilizada uma analogia entre a hierarquia de memória e

processador, em três partes, *tag* (ou rótulo), *index* (ou índice) e *offset* [1]. A informação que é apontada pelos $M+1$ *bits* menos significativos do endereço, é chamada *offset*, e é utilizada apenas para saber qual *byte* do bloco o processador está tentando acessar (a quantidade total de blocos é obtida com a expressão 2^{M+1}).

Figura 2 – Exemplo de um controlador de memória cache

As linhas da memória cache são definidas pelo intervalo entre o *bit* N e M, de acordo com a quantidade de linhas (ou conjuntos se for do tipo associativa) que a memória cache possuir, onde 2^{N-M} é o número de linhas. Este intervalo é chamado

index, e irá variar de acordo com a quantidade de linhas que a cache possuir.

Por fim, o intervalo restante, definido pelos *bits* mais significativos, partindo do *bit* N, é chamado de *tag*. Este intervalo é utilizado para reconhecer a qual setor da memória principal pertencem os dados do bloco em uma linha da memória cache.

A Figura 2 representa um controlador de uma cache de 64KB, com blocos de 32 *bytes* e arquitetura de 32 *bits* e mapeamento direto, onde se pode observar a ação do índice, do rótulo e do offset no controle de dados da memória cache.

2.1.4 Associatividade

Algumas técnicas podem ser utilizadas para uma melhor utilização da memória cache de acordo com as necessidades encontradas. Uma destas técnicas é a associatividade, que é a divisão da memória cache em um ou mais conjuntos de linhas.

Existem três tipos de associatividade, o mapeamento direto que é conhecido por possuir o melhor tempo de resposta dentre as associatividades apesar de possuir um desempenho inferior aos outros modelos, a memória cache totalmente associativa que é o modelo ótimo dentre as associatividades e com a união desses dois modelos básico de associatividade, chegou-se a um terceiro modelo de associatividade, a associação por conjuntos, um dos modelos mais utilizados em processadores na atualidade.

2.1.4.1 Mapeamento direto

Dentre os modelos de associatividade, o mapeamento direto aparece como o mais simples de ser implementado. Nesta associatividade, cada bloco de endereço de memória pode ser adicionado em apenas uma linha da memória cache. As linhas estão diretamente associadas ao índice, e para saber a qual parte da memória o bloco pertence, o *tag* do endereçamento da memória é copiado ao *tag* da linha do cache utilizada.

Quando o processador for acessar a memória, consulta a linha pertencente ao *index* da memória, e compara os *tags* para saber se o dado contido na linha é o dado necessário. Em cache *miss*, a linha é sempre substituída. A Figura 3 demonstra a associação dos blocos de memória com as linhas da memória cache, onde pode ser observado que cada bloco de memória possui apenas uma linha

possível de ser armazenado na cache.

Figura 3 – Mapeamento direto

2.1.4.2 Mapeamento totalmente associativo

Neste tipo de mapeamento, os blocos de endereço da memória principal acessados podem ser copiados em qualquer linha da cache livre. O *tag* e o *index* neste caso são um só dado, que serão copiados no *tag* da linha da cache para saber a que parte da memória principal aquela linha pertence. Durante uma consulta à memória, o controlador de cache compara todos os *tags* da cache simultaneamente, para saber se o dado está ou não contido na memória.

Em cache *miss*, o bloco de memória obtido é adicionado em qualquer linha livre e, não havendo linhas livres, algum algoritmo de substituição será utilizado para a substituição da linha. Este tipo de associatividade não é utilizada pois possui um sistema muito complexo de comparação ao ser implementado.

A Figura 4 representa a associação dos blocos da memória com as linhas da memória cache. Como pode ser observado, um bloco de memória pode ser adicionado a qualquer linha livre dentro da cache.

Figura 4 – Mapeamento totalmente associativo

2.1.4.3 Mapeamento associativo por conjunto

Neste tipo de mapeamento divide-se todas as linhas do cache em conjuntos com 2^K linhas. Entre si, os conjuntos seguem o mapeamento direto, usando o índice para saber a que conjunto cada bloco de memória pertence. Internamente, os conjuntos são totalmente associativos, possuindo linhas com o mesmo índice, contudo com rótulos diferentes. Esta é a associatividade mais comum utilizada atualmente e não possui o mesmo problema da totalmente associativa pois, normalmente, os conjuntos possuem algo em torno de quatro, oito ou no máximo dezesseis linhas, sendo muito mais simples implementar um sistema de comparação dos rótulos simultaneamente.

Figura 5 – Mapeamento Associativo por conjuntos

A Figura 5 representa a associação dos blocos da memória com os conjuntos de linhas da memória cache, onde cada bloco de memória pode ser adicionado a apenas um conjunto, e internamente ao conjunto, este bloco pode ser adicionado em qualquer linha livre dentro da cache.

2.1.5 Políticas da memória cache

Após a definição do tipo de associatividade que irá ser utilizado, é necessário que sejam definidas as políticas de gerência da cache, que são divididas em duas categorias, a política de substituição, que basicamente define as regras a serem aplicadas durante a substituição de uma linha na cache, e a política de escrita, que define qual estratégia será utilizada quando for atualizar um dado novo na memória.

2.1.5.1 Algoritmos de substituição

O algoritmo de substituição está entre uma das possíveis políticas de funcionamento de uma memória cache. Ele está diretamente ligado às memórias cache associativas por conjunto, sendo sua função analisar qual das linhas de determinado conjunto (ou do único conjunto, no caso específico das memórias totalmente associativas) deverá ser substituída quando ocorrer uma falta neste conjunto. Dentre os possíveis algoritmos de substituição a ser utilizado, estão [1][3]:

- **FIFO** (*first in first out* – primeiro a entrar primeiro a sair) - funciona como uma fila onde a primeira linha a ser adicionada em um conjunto da memória, será a primeira a ser substituída quando necessário. Sua implementação em associações 2-way pode ser feita utilizando um *bit* que será setado para a linha carregada anteriormente toda vez que uma linha for carregada, substituindo a linha que estava com o *bit* setado. Para associações maiores, pode ser utilizado um contador para cada linha, onde o valor mais baixo será a última linha que foi adicionada, e o valor mais alto, a primeira linha;
- **LRU** (*least recently used* – menos usado recentemente) – funciona como o FIFO, porém, toda vez que ocorre um cache *hit* com uma linha que já está presente na fila, esta linha volta para o início da fila. Usando este algoritmo em uma associação 2-way é parecido com o FIFO, porém o *bit* será usado para informar qual a linha mais recente do conjunto. Uma proposta feita pela Intel em associações maiores é usando o algoritmo pseudo-LRU. Em uma

associação 4-way, três *bits* são utilizados. O primeiro *bit* será setado se ocorreu um *hit* nas linhas 1 e 2, e zerado se ocorreu um *hit* nas linhas 3 e 4. O segundo *bit* será setado se ocorreu um *hit* em 1 e zerado se ocorreu *hit* em 2, e a mesma coisa para o terceiro *bit* nas linhas 3 e 4. A pequena diferença entre esta implementação e a real pode ser demonstrada em um *loop*, onde a CPU fica acessando a seqüência 1, 2, 3, 1, 2, 3, e o 4 seria provavelmente a linha a ser substituída, porém, com esta seqüência, os *bits* estarão com o valor “001”, ou seja, a linha a ser substituída será a linha 1. As chances deste erro ocorrer são estatisticamente baixas;

- MRU (*most recently used* – mais usado recentemente) – funciona como o LRU, mas quando uma substituição for necessária, substitui-se a primeira da fila, a linha que foi utilizada mais recentemente. A implementação pode ser feita usando um *bit* para informa qual é a linha que foi acessada mais recentemente para ser substituída;
- LFU (*least frequently used* – menos usado freqüentemente) – apesar deste algoritmo não ser muito utilizado em projetos de cache, ele tem sua importância em algoritmos de substituição de páginas, nos sistemas operacionais. Pode ser implantado utilizando um contador em cada linha e um ponteiro por conjunto. Quando ocorre um *hit*, e o ponteiro do conjunto estiver apontando para a linha em que ocorreu o *hit*, o contador desta é incrementado, caso contrário, o ponteiro aponta para a linha em que ocorreu o *hit*. Quando ocorrer um *miss*, a linha do conjunto que tiver o menor valor no contador será a utilizada na substituição, e os contadores serão zerados para que esta linha não seja substituída logo no primeiro *miss* que ocorrer;
- NLU (*not last used* – não usado por último) – o funcionamento deste algoritmo é uma mistura do LFU com uma substituição aleatória. A idéia parte do princípio que a substituição aleatória é interessante, porém, normalmente não é tão interessante substituir a última linha acessada. Para a implementação basta adicionar um ponteiro para a última linha acessada dentro de um conjunto, e na hora que houver um *miss*, substitui-se aleatoriamente uma das linhas, exceto a linha que está apontada pelo ponteiro.

2.1.5.2 Estratégias de escrita

Quando o processador for atualizar um dado que está na memória cache para a memória RAM duas estratégias diferentes podem ser utilizadas, *write-back* ou *write-through*. Na *write-back*, o processador atualiza apenas o dado da memória cache, e quando este dado for ser substituído por outro, o controlador da cache faz a atualização do dado modificado na memória principal. Esta estratégia é muito usada por ser mais rápida, já que a memória será acessada menos frequentemente.

Na *write-through*, sempre que alguma atualização é feita em um dado contido na cache, esta atualização também será feita na memória. Esta estratégia é mais lenta, pois a memória principal sempre será acessada quando uma modificação for feita, porém, ela é mais simples de ser implementada e mantém a consistência, muito útil para sistemas multiprocessados.

2.1.6 Memórias cache unificadas e não-unificadas

Outra característica que as memórias cache podem possuir é a possibilidade da memória de dados e da memória de instruções não serem unificadas. As memórias caches não unificadas foram propostas na arquitetura Harvard, utilizadas pela primeira vez no computador Mark I [21]. Esta arquitetura é uma evolução da arquitetura Princeton (ou Von Neumann, como é conhecida) [22].

Figura 6 – Arquitetura Von-Neumann vs. Arquitetura Harvard

A principal característica que diferencia as duas arquiteturas citadas está na organização das memórias. A arquitetura de Von-Neumann caracteriza-se pela

unificação da memória de dados e instruções em um só barramento, enquanto a arquitetura Harvard divide o barramento de acesso à memória em memória de dados e memória de instruções [23], como pode ser observado na Figura 6.

Processadores utilizando a arquitetura Harvard possuem as memórias cache de dados e instruções separadas, pois o acesso a essas memórias é feito em barramentos diferentes.

A utilização da arquitetura Harvard em processadores modernos é cada vez maior entre a grande maioria dos desenvolvedores de processadores de uso geral, como a Intel, que está adotando esta arquitetura desde os processadores da família Pentium, até os atuais Core 2 Quad [24].

Um dos grandes avanços com esta arquitetura está na melhoria total de desempenho do sistema. Os processadores atuais utilizam *pipelines* na tentativa de simular um sistema paralelo, e com isso, é interessante que, enquanto o processador estiver buscando novas instruções, ele também possa estar copiando para a memória os resultados das últimas instruções executadas e isso só é possível com o uso de barramentos separados para o acesso das memórias de dados e de instruções. Outro fator que torna essa arquitetura muito útil é o fato de muitos sistemas utilizarem uma memória EEPROM ou até mesmo FLASH somente para a parte de instruções enquanto a memória RAM é utilizada principalmente com os dados. Este caso é amplamente utilizado em sistemas embarcados, como os atuais telefones celulares.

2.1.7 Memórias cache multi-níveis

Ao analisar o desempenho de um computador seria interessante que não houvesse latência nos acessos à memória, de modo a se imaginar o sistema como sendo de acesso instantâneo, ou seja, o dado de um certo endereço de memória será obtido no exato momento que ele for acessado. Infelizmente esta não é a realidade, sendo esse um dos motivos da utilização de memórias cache [3].

As memórias cache são utilizadas junto ao processador, por serem memórias mais rápidas, o que causa ao processador a impressão de um acesso quase instantâneo ao dado solicitado. Mas as memórias cache não comportam todos os dados utilizados pelo processador durante a execução dos programas, principalmente pelo tamanho das memórias cache comparadas ao tamanho do fluxo

de dados do uso normal de um computador.

Dado este problema, por que não utilizar uma memória cache maior, dez vezes maior, cem vezes maior, com a possibilidade de armazenar muito mais dados junto ao processador? A resposta para esta dúvida está no fato que a tecnologia das memórias cache utilizadas é muito cara e o ganho de desempenho que o processador obtém com esses aumentos não compensa o gasto total, tornando economicamente inviável um aumento muito grande nessas memórias.

Figura 7 – Hierarquia de memórias

A Figura 7 mostra uma hierarquia de memória, onde pode ser observada a relação que a velocidade de acesso de um nível de memória tem com o custo de fabricação deste nível, que é inversamente proporcional ao tamanho que este nível pode possuir e com a proximidade dele à CPU. Por exemplo, os registradores são memórias muito velozes e próximas à CPU. Pelo seu custo muito elevado, possuem uma quantidade total de memória deste tipo muito escassa.

Para solucionar o problema da escassez das memórias cache sem aumentar demais o custo total, foi proposta a inclusão de uma memória cache de menor velocidade de acesso, portanto, com um custo menor, entre a memória cache L1 e a memória RAM [25].

O modo de funcionamento deste nível de memória é basicamente parecido com o L1. No L1, quando ocorre uma falta, o processador busca o dado do endereço de memória diretamente da memória RAM, mas para este caso, quando ocorre uma

falta na L1, ao invés de fazer a busca diretamente na memória RAM, o processador faz primeiro a busca na memória cache L2, com uma latência maior que a L1, porém ainda muito menor que a memória principal. Apenas no caso de faltas na L2 o processador faz a busca do dado na memória RAM principal [1].

Os processadores podem possuir, ainda, outro nível de memória cache, o L3. Alguns processadores atuais possuem este terceiro nível com a latência de acesso maior que o L1 e o L2, mas ainda menor que a memória principal. Alguns dos processadores da família x86 da Intel possuem este terceiro nível de memória cache, como, por exemplo, o Pentium IV Extreme Edition de 0,13 microns [26].

2.2 Computação Reconfigurável

Na computação moderna existem basicamente dois métodos de se executar um dado algoritmo [6], a execução em *hardware*³ e em *software*⁴. O método em *hardware* é um modo muito rápido e eficiente de se realizar a execução dos algoritmos, porém eles contam com um grande problema, não podem ser alterados após a sua fabricação, sendo assim, a modificação deste sistema após a sua fabricação para realizar um algoritmo diferente acaba se tornando uma coisa difícil pois é necessário se fabricar um outro *hardware*, o que comprova que este não é um sistema flexível.

A execução em *software*, por outro lado, é um método altamente flexível de se executar algoritmos, pois é facilmente modificada já que para realizar um dado algoritmo ele executa um conjunto de instruções, sendo assim um sistema muito mais volátil que o *hardware*, pois facilmente pode se adaptar a diferentes finalidades. Porém, o *software* possui um desempenho final bastante inferior ao desempenho obtido em sistemas de *hardware* fixo, pois para realizar determinada instrução ele precisa ler a memória em busca da instrução, decodificá-la para então poder executá-la, e durante um algoritmo esta operação é realizada diversas vezes.

A computação reconfigurável apareceu para preencher o espaço entre o *hardware* e o *software*, obtendo um sistema híbrido que pudesse obter performances potencialmente maiores que as obtidas em *software*, mantendo um nível de flexibilidade bem maior que os obtidos em *hardwares* de lógica fixa.

³ Conhecidos como hardware de lógica fixa e largamente representados pelos circuitos integrados de aplicações específicas, ou ASICs [11-a].

⁴ Combinação de software e microprocessador [11-a].

2.2.1 Conceituação

Ao mencionar computação reconfigurável, os autores de [5] buscam a definição e diferenciação de alguns conceitos importantes e outros aspectos relacionados ao assunto, tais como:

- programação ou configuração são os conceitos mais complicados, mas um modo simples para seu entendimento é que programação está ligada diretamente ao ato de programar o dispositivo a realizar certas operações (programas em um microprocessador) e configuração está ligada diretamente ao ato de configurar um dispositivo, como a forma que os objetos são conectados internamente (roteamento em uma FPGA);
- configuração ou reconfiguração estão diretamente relacionadas com a quantidade de configurações que um dispositivo pode realizar;
- reconfiguração estática é a reconfiguração realizada no estado inativo do dispositivo e reconfiguração dinâmica é a realização da reconfiguração durante o seu funcionamento normal [27];
- reconfiguração parcial é a possibilidade de reconfigurar apenas alguns dos recursos do dispositivo e reconfiguração total é a possibilidade de reconfigurar todos os recursos do dispositivo;
- contexto de reconfiguração único ou múltiplo está relacionado com a existência de um ou mais padrões de reconfiguração no dispositivo;
- reconfiguração em tempo de compilação é a reconfiguração realizada apenas no início da fase de execução relacionada normalmente com a compilação e a reconfiguração em tempo de execução está relacionada com a reconfiguração do dispositivo durante a execução;
- reconfiguração local ou remota está relacionada com a proximidade física, ou não, do padrão de configuração do dispositivo;
- reconfiguração normal ou rápida está diretamente ligado ao tempo necessário para a realização da reconfiguração;
- cache de reconfiguração está relacionado com a utilização de técnicas de cache para armazenar *bits* de reconfiguração.

2.2.2 Dispositivos Reconfiguráveis

Os dispositivos de lógica programável (PLD)⁵ vêm sendo largamente utilizados e em muitos casos nem são reconhecidos como tais. Estes dispositivos têm a característica de implementar diversas funções lógicas sem necessariamente executá-las.

O caso mais primitivo de um PLD é o uso de EPROMs⁶ e, posteriormente, EEPROMs⁷ para a implementação de funções lógicas [28]. Apesar de serem fabricadas como memórias, estes dispositivos são facilmente utilizados com dispositivos programáveis utilizando os pinos de seleção de endereços de memória como entradas lógicas e os dados nos endereços de memória como saídas das possíveis combinações lógicas implementadas neste dispositivo.

Em 1970, a Texas Instruments lança um dispositivo de lógica programável com até 17 portas de entradas e 18 portas de saída e 8 flip-flops JK para memória. O TMS2000 é chamado de PLA (*Programmable Logic Array*) pela própria TI. Após esta denominação, outros produtos são lançados utilizando esta denominação, como o DM7575 da National Semiconductor, lançado em 1973, com 14 pinos de entrada e 8 de saída, e sem nenhum registrador de memória.

Outros conceitos de PLD aparecem posteriormente. A MMI lança o conceito de PAL (*Programmable Array Logic*) e a Lattice Semiconductor lança as primeiras GAL, também conhecidos exemplos de dispositivos de lógica programável.

Com a necessidade de utilização de lógicas mais complexas, surgem, finalmente, dispositivos como os CPLDs⁸ e as FPGAs⁹, muito utilizados na atualidade. Dentre as empresas mais influentes na fabricação destes dispositivos estão a Xilinx [29] e a Altera [30]. Como maior destaque entre os CPLDs estão a família CoolRunner e o CoolRunner II da Xilinx, e a família MAX da Altera, e entre as FPGAs de maior destaque estão a família Virtex e a família Spartan da Xilinx, e a família Apex, a família Flex e a família Stratix da Altera.

A interface com estes dispositivos normalmente é feita através de programas CAD como os Max+Plus II e Quartus II da Altera e o ISE Foundation da Xilinx, e

⁵ Programmable Logic Device

⁶ Erasable Programmable Read Only Memory

⁷ Electrically Erasable Programmable Read Only Memory

⁸ Complex Programmable Logic Devices

⁹ Field Programmable Gate Array

ambas as empresas possuem a possibilidade de se desenvolver as aplicações tanto usando projeto esquemático ou usando linguagens de descrição de *hardware* como a VHDL e o Verilog.

2.2.3 Processadores e a Reconfiguração

Na busca do aumento no desempenho em operações específicas, o uso da reconfiguração em processadores vem aumentando. Estudos vêm sendo realizados aplicando a reconfiguração nas mais diversas áreas do processador, como o conjunto de instruções, onde a reconfiguração age em conjuntos de instruções de propósito específico a aplicações, em paralelo ao conjunto de instruções de uso geral.

O PRISM-I [31]¹⁰ e seu sucessor, PRISM-II [32], são processadores com a configuração em seu conjunto de instruções. Este sistema é separado em dois nodos interligados por um barramento, sendo um nodo de processamento não reconfigurável de uso geral baseado em processadores comerciais, e o outro nodo utilizando FPGAs para a execução de instruções específicas, como por exemplo, a realização de logaritmos na base 2 e algoritmos de correção de erro.

O Chimaera [33], um processador que realiza reconfiguração na unidade funcional, propõe a inclusão de uma matriz reconfigurável junto ao processador com a intenção de aumentar o desempenho total do sistema com o processador fixo e a parte reconfigurável integradas. Um dado interessante do Chimaera é que ele trata a lógica reconfigurável como um cache de instruções reconfiguráveis, onde as instruções mais utilizadas atualmente estão carregadas, e sempre que for necessário carregar uma função que não esteja lá, esta instrução é carregada e, se preciso, substitui uma ou mais funções não usadas há algum tempo.

Outro processador que trabalha com reconfiguração é o Garp [34], que propõe a utilização de um processador MIPS-II em conjunto com um co-processador reconfigurável. A principal característica do co-processador reconfigurável nesta arquitetura é a execução de laços de aplicações liberando o processador para continuar executando outros trechos de código.

Muitos outros trabalhos na área acadêmica podem ser citados com a utilização de reconfiguração junto ao processador, como o OneChip [35] que

¹⁰ Processor Reconfiguration through Instruction-Set Metamorphosis

trabalha com uma unidade funcional reconfigurável em paralelo a unidade funcional fixa e seu sucessor, OneChip-98, que buscando uma melhora optou pela adiação de caches que poderiam ser utilizados tanto pela unidade reconfigurável quanto pela fixa. O PipeRench [36], que utiliza uma forma de *pipeline* de configurações, o Morphosys [37], que utiliza uma malha reconfigurável junto ao processador para explorar o paralelismo das aplicações, e o DISC [38]¹¹ são outros exemplos interessantes do estudo da utilização de reconfiguração em processadores.

Não é apenas a área acadêmica que vem investindo neste conceito, alguns fabricantes de supercomputadores começaram a lançar computadores com FPGAs embutidas. Incorporando ao sistema uma FPGA Virtex II Pro da Xilinx, a Cray lançou o XD1. Esta arquitetura reconfigurável se mostrou uma ótima solução para implementação de funções de busca, processamento de sinais, manipulação de áudio, vídeo e imagem, criptografia, codificação e decodificação, correções de erro e geração de números aleatórios para simulações de Monte Carlo, além de também poder ser utilizada em outras funções como processamento sísmico, aceleração gráfica, física quântica, simulação de tráfego veicular, entre outras [39].

Outras empresas que vêm utilizando esta técnica são a SGI, com o RASC [40], e a RSC Computers, que lançou máquinas usando a sua tecnologia IMPLICIT+EXPLICIT, onde *implicit* é basicamente composto por *hardware* de lógica fixa como ASICs e DSPs, e *explicit* é composto por *hardware* de lógica reconfiguráveis como as FPGAs [41].

Estes são apenas alguns dos possíveis usos da reconfiguração aplicável à computação, que faz com que um sistema possa melhor se adaptar aos seus atuais requisitos e obter um melhor desempenho [5]. Mas a idéia de reconfiguração em processadores não está apenas na utilização de lógicas reconfiguráveis na unidade funcional e no conjunto de instruções em busca de uma melhora de desempenho na execução de certas aplicações, e sim também em uma melhor adaptação do sistema às características de uso, em busca de um desempenho melhor. A partir desta idéia, diversos trabalhos vêm sendo realizados com a idéia de tentar implementar reconfigurabilidade às memórias cache, com intuito principal de diminuir a taxa de faltas, aumentando assim o desempenho total do sistema.

¹¹

2.3 Cache Reconfigurável

Como já visto, a diminuição do gargalo criado entre memória e processador é uma das maiores preocupações dos desenvolvedores de processadores. A implementação de caches, porém, não vem dando conta de realizar sozinha este serviço, e alguns trabalhos estão sendo realizados para aumentar o desempenho das memórias cache, diminuindo o número de faltas, como a implementação de caches *victim* [42], que basicamente é a introdução de linhas suplementares entre o *L1* da cache e o nível subsequente que armazenaria as linhas recentemente descartadas pela cache durante um *cache miss*.

Mas a implementação de técnicas simples como esta não são suficientes, portanto nada mais normal que a adaptação de lógicas de reconfiguração também nas hierarquias de memória.

2.3.1 Reconfigurabilidade em memórias caches

Diversos trabalhos foram realizados com memórias cache reconfiguráveis, executando a reconfiguração nos diferentes campos de uma memória cache. Estes trabalhos propõem diversas técnicas para alterar o comportamento de uma cache durante seu modo normal de operação.

A D-NUCA [43]¹² é uma proposta que utiliza uma arquitetura de memória diferente do atuais ML-UCA¹³, onde a memória seria dividida em vários bancos e os bancos mais próximos iriam ter tempo de acesso menor que os mais distantes. Na proposta inicial de uma NUCA, os dados só poderiam ser adicionados a certos bancos de memória, de acordo com seu *tag*. A nova proposta, D-NUCA, trabalha com a reconfiguração na alocação de dados nos diversos bancos, de acordo com a frequência com que são solicitados.

Esta proposta de reconfiguração em memórias cache é muito interessante, não somente pela proposta de uma nova arquitetura de caches, mas também pela utilização de reconfiguração nesta proposta. Porém, a maioria das propostas nessa área utilizam o conceito popular de memórias cache e tentam realizar a reconfiguração dentre os diversos campos que a cache possui.

Dentre os possíveis campos que podem ser reconfigurados está o tamanho

¹² Dynamic Non-Uniform Cache Architecture

¹³ O autor nomeia a organização atual de Multi-Level Uniform Cache Architecture

da linha, a exemplo do trabalho proposto por Veidenbaum e outros [44]. O conceito desta proposta é basicamente a implementação de uma grande memória cache, provida de linhas com blocos 8 *bytes* cada e utilizando mapeamento direto. Esta arquitetura de memória cache realiza a sua reconfiguração formando diversas linhas virtuais, que podem usar uma ou mais linhas concatenadas, variando sempre na potência de 2, com um mínimo de 8 *bytes* e o máximo de 256 *bytes*. Sua arquitetura se baseia no princípio de que duas linhas adjacentes que possuam o mesmo *tag* (rótulo) possuem uma boa localidade espacial, podendo assim, ser criada uma linha unificada com as duas linhas concatenadas em uma só. Outro princípio desta arquitetura é que ao substituir uma linha, se for observado que metade dela não está sendo utilizada, ou esta linha não possui uma boa localidade espacial, ou ela está em conflito com outra linha. Sendo assim, a linha será particionada em duas linhas distintas para evitar poluição do cache com dados buscados da memória.

Outro campo de pesquisa para memórias cache reconfiguráveis é a política de substituição de linhas, onde dois trabalhos se destacam. O SF-LRU [45], que é uma variação do LRU. O algoritmo é dividido em duas partes distintas de operação, a operação de leitura da memória e a operação de escrita na memória. Ambas as operações utilizam um registrador que controla a frequência de utilização, chamado RFCV (*Recency-Frequency Control Value*). Durante a operação de escrita, uma análise é feita sobre o conteúdo do RFCV do último dado da fila com o conteúdo do RFCV de seu antecessor, e se o conteúdo dele for maior, seu antecessor deverá ser eliminado e o conteúdo de todos os RFCV devem ser reiniciados (o algoritmo fornece apenas uma outra chance dos dados mais utilizados de permanecerem na cache), e o dado novo a ser escrito entra no topo da fila, como no LRU. Na operação de leitura, o conteúdo do RFCV do dado lido é incrementado, e o dado recém lido vai para o início da fila, assim como o LRU.

Um outro projeto na política de substituição de linhas utilizou o que se chamou de técnicas adaptativas [46]. A arquitetura deste trabalho não tem como base nenhum dos algoritmos de substituição conhecidos. Sua implementação é obtida com o uso de uma fórmula definida pelo autor que ajuda a identificar qual dos blocos deverá ser substituído. Esta fórmula leva em consideração a frequência de acessos à linha e o número de *hits* (acertos) ocorridos para obter um resultado que será comparado a um registrador α que define se a memória cache no momento atual estará dando prioridade ao número de *hits* ou ao tempo entre um acesso e

outro a uma mesma linha. Além da implementação da fórmula, serão necessários alguns *bits* por linha e registradores a mais para a implementação desta arquitetura. A técnica adaptativa neste algoritmo está na mudança do valor do registrador α com a carga de trabalho do sistema.

2.3.2 Reconfiguração na Associatividade

Um outro campo na reconfiguração em caches é a reconfiguração na associatividade. Um cache muito interessante que utiliza este conceito é o cache V-Way [47]. Este cache parte do conceito que as aplicações apresentam uma distribuição não uniforme de acessos à memória, por isso muitas vezes pode ocorrer de haver conjuntos super-utilizados enquanto outros conjuntos podem ter linhas de sobra, o que deixa em aberto a idéia do primeiro conjunto utilizar a linha sobrando do segundo conjunto. A implementação deste cache é realizada com a duplicação dos campos de *tag*, com isso, haveria o dobro de campos *tag* do que linhas de dados. Outra diferença é que não haveria mais um mapeamento direto entre um *tag* e uma linha de dados, ao invés disso os *tags* iriam conter um campo para armazenar um ponteiro para uma das linhas de dados.

2.3.2.1 Algoritmo Carvalho e Martins

O algoritmo Carvalho e Martins [48]¹⁴, um algoritmo que realiza reconfiguração na associatividade, se destaca com sua simplicidade e facilidade de aplicar em *hardware*. Esta arquitetura de cache reconfigurável utiliza uma política de re-adaptação bem simples. A partir de uma memória cache inicialmente definida, a memória cache vai se readaptando à carga de trabalho utilizando intervalos de tempo pré-definidos, em busca da configuração ideal.

O modo de funcionamento é bastante simples, utilizando-se inicialmente uma cache normal por associação em conjuntos com dois registradores em cada conjunto, um fará o registro dos acessos ao conjunto e o outro o registro das faltas. Estes registros serão realizados no que iremos chamar de modo normal de funcionamento. Existem dois modos distintos de funcionamento, o modo normal, referido anteriormente, onde esta memória cache funcionará basicamente como uma

¹⁴ Como o algoritmo proposto em [48] não foi nomeado pelos autores, iremos nos referir a ele por “algoritmo Carvalho e Martins”, utilizando o sobrenome dos autores do trabalho.

memória cache comum, e o modo de reconfiguração a ser chamado em intervalos de tempos pré-determinados, onde será executado o algoritmo com sua política de adaptação.

O algoritmo de adaptação inicialmente calcula a média de acessos e a média de faltas de todos os seus conjuntos, definindo assim, uma barreira para classificar os conjuntos pela sua quantidade de acessos e de faltas. Definidas as suas barreiras, cada um dos conjuntos irá receber um rótulo, a partir de sua classificação de acordo com o número de acessos e de faltas. Os possíveis rótulos, a serem dados aos conjuntos, definidos na Tabela 1.

Tabela 1 – Descrição de rótulos

Tipo	Faltas	Acessos	Descrição
GG	Grande	Grande	Tem alta prioridade em receber linha
GP	Grande	Pequeno	Tem baixa prioridade em receber linha
PG	Pequeno	Grande	Conjunto estável, não doa e não recebe
PP	Pequeno	Pequeno	Conjunto doador de linha

Os rótulos funcionam para definir a ação a ser tomada durante a adaptação da memória cache. Os conjuntos podem ser rotulados como GG, GP, PG e PP de acordo com sua posição nas barreiras de faltas e acessos.

Os conjuntos rotulados como GG obtiveram uma grande quantidade de acessos e uma grande quantidade de faltas durante o intervalo de tempo analisado. Com essas características, ele se torna um conjunto com alta prioridade para receber uma linha extra.

Os conjuntos GP obtiveram uma grande quantidade de faltas e uma baixa quantidade de acessos durante o intervalo de tempo medido, portanto, eles possuem características de um receptor, por haver uma alta quantia de faltas, mas por seus acessos serem baixos, têm uma baixa prioridade para receber uma linha nova.

Os conjuntos classificados como PG são os conjuntos ideais, pois possuem baixa quantidade de faltas e uma alta quantidade de acessos. Estes são considerados como conjuntos que não precisam receber linhas novas e nem doar linhas, apresentando um equilíbrio com relação ao número de faltas e acessos.

Por fim, conjuntos definidos como PP são os reconhecidos como doadores, por

possuir uma baixa quantidade de acessos durante o intervalo de tempo medido, e também por possuir uma baixa quantidade de faltas, sendo doadores potenciais.

Tabela 2 – Exemplo de classificação

Conjunto	Faltas	Acessos	Rótulo
S0	10	100	GG
S1	7	30	PP
S2	15	110	GG
S3	8	40	PP
S4	8	50	PP
S5	12	20	GP
S6	3	120	PG
S7	17	100	GG
Média	10	70	

A Tabela 2 mostra um exemplo funcional de uma classificação de conjuntos. Neste exemplo, uma cache fictícia com 8 conjuntos (S0, S1, S2, ..., S7) tem suas estatísticas analisadas durante um intervalo de tempo. Essas estatísticas que são descritas pelos valores de seus registradores, definidos pela coluna de faltas e acessos da tabela. A média de acessos e faltas é calculada pela soma de todos os registradores dos conjuntos dividido pelo número total de conjuntos. A partir das médias das barreiras, os rótulos dos conjuntos são definidos, analisando sua posição em relação ao número de faltas e acessos.

Após a obtenção das classificações dos conjuntos, o algoritmo de adaptação tem a função de realizar as doações propriamente ditas. Nesta fase, apenas três restrições são impostas aos conjuntos: um conjunto com apenas uma linha não poderá doar a sua última linha, pois os conjuntos não poderão deixar de existir, mudando a arquitetura final do sistema; a arquitetura dos sistemas será definida por *x-y way*, onde o valor de *x* é o valor da associatividade inicial, e o valor de *y* é o valor da associatividade máxima, portanto, um conjunto que possui *y* linhas, não poderá receber mais nenhuma linha, pois esse valor *y* é o número de comparadores que o sistema possui; um conjunto pode fornecer apenas uma linha por vez, e o mesmo acontece com os conjuntos receptores, que podem receber apenas uma linha por rodada de adaptação.

Dada uma rodada de adaptação (execução do algoritmo de adaptação), o sistema voltará ao seu modo normal de funcionamento, até a próxima adaptação. Ao finalizar o algoritmo, os valores dos registradores de faltas e acessos, de todos os

conjuntos, serão zerados para obter estatísticas novas, referentes ao novo intervalo de operação.

A duração do modo normal de funcionamento é definida por um intervalo de tempo pré-determinado. Este intervalo de tempo não pode ser relativamente baixo, para poder obter uma estatística confiável, e nem muito grande, para que a função de readaptação seja chamada um número de vezes suficiente para obter uma melhoria do sistema.

A implementação deste sistema ocorre de uma forma bem simples, sendo necessário apenas dois registradores adicionais por linha para obter o número de acessos e de faltas por conjunto, e de um número de comparadores suficientes para analisar o valor máximo de linhas que cada conjunto poderá obter. A média dos acessos e das faltas pode ser calculada facilmente usando um somador de todos os registradores. Como o número de conjuntos é uma potência de 2, a operação de divisão pode ser realizada apenas com um deslocador.

A grande vantagem deste sistema é que ele apresenta ganhos de desempenho (aumento da taxa de acertos) bastante significativos em relação aos sistemas não adaptativos. Além disso, o modo simplificado de funcionamento e sua arquitetura fazem que o *hardware* exigido para a implementação desta memória cache seja bem elementar.

3 PROPOSTA DE RECONFIGURAÇÃO DA ASSOCIATIVIDADE

Neste capítulo serão apresentadas as principais deficiências das propostas atuais, bem como a proposta de um novo algoritmo de reconfiguração de associatividade em memórias cache que busca suprir as deficiências detectadas nas propostas estudadas.

3.1 Deficiências nas propostas atuais

Ao estudar o algoritmo Carvalho e Martins, foi possível observar duas grandes desvantagens em sua política de reconfiguração. Primeiro, em relação à classificação de conjuntos pelo número de faltas e acessos, pelo fato de dividir os conjuntos em apenas dois grupos, acaba colocando na mesma classificação conjuntos com características bastante distintas. No exemplo observado anteriormente na Tabela 2, os conjuntos S0 e S7 são classificados como GG, então têm a mesma prioridade para receber uma linha. Desta forma é bem provável que o conjunto S0 receba uma linha antes que o conjunto S7. Contudo, o conjunto S7 deveria ter uma prioridade maior que o conjunto S0, por possuir um número de faltas bem mais elevado.

Uma outra situação desfavorável para o algoritmo Carvalho e Martins ocorre quando a carga de trabalho apresenta um padrão de acessos em que um ou mais conjuntos alteram seu padrão doador-receptor, isto é, um conjunto passa, por exemplo, de doador de linhas para receptor durante a execução ou vice-versa. Esta alteração de padrão pode ser explicada talvez pelo fato das diversas cargas de trabalho aplicadas às memórias cache não possuírem uma distribuição uniforme de acessos. Como o algoritmo permite apenas a doação de uma linha por vez, a transição do padrão doador-receptor será mais lenta. Este fato faz com que o algoritmo deixe de alcançar melhores ganhos de desempenho.

Trabalhando nestes dois aspectos, é possível obter um algoritmo que produza uma cache com melhor desempenho.

3.2 Proposta do novo algoritmo

Nossa proposta de um algoritmo de reconfiguração de memória cache

trabalha realizando também a reconfiguração da associatividade da cache. Estamos propondo uma evolução do algoritmo Carvalho e Martins, mas com as melhorias necessárias para fazer que o sistema seja mais criterioso ao definir as prioridades dos conjuntos analisados [18]¹⁵.

Em uma memória cache comum, a sua configuração é mantida após a sua fabricação e dependendo da carga de trabalho aplicada a esta memória cache ela pode possuir resultados insatisfatórios por causa dos pontos apresentados na seção anterior. Nossa proposta busca uma arquitetura de cache que possua uma configuração inicial igual as memórias cache comuns, mas com a possibilidade de se reconfigurar utilizando uma política de reconfiguração composta de nosso algoritmo adaptativo que irá realizar a re-adaptação de acordo com a carga de trabalho aplicada a memória cache como representa a Figura 8.

Figura 8 – Arquitetura de cache reconfigurável

A memória cache que foi utilizada parte inicialmente de uma memória cache com associatividade maior que um, caso contrário tornaria o sistema incapaz de realizar reconfiguração, já que utilizou-se como definição de projeto que nenhum conjunto de nossa memória cache pode ficar sem linha alguma. A cache também não pode possuir uma associatividade máxima muito alta, o que tornaria este sistema inviável de se implementar em *hardware* pela complexidade do sistema de comparação.

Uma arquitetura de memória cache é composta de tantas linhas quanto forem possíveis serem obtidas de acordo com o seu tamanho total e o tamanho do blocos de dados obtidos da memória, Este valor pode ser obtido com a equação 3.1.

¹⁵ Iremos nos referir ao algoritmo proposto como algoritmo “Kerr e Midorikawa”, utilizando o sobrenome dos autores.

$$NúmeroLinhas = \frac{TamanhoTotal(KB)}{TamanhoBloco(bytes)} \quad (3.1)$$

A partir do número de linhas que uma memória cache possui e a sua associatividade, pode se obter a quantidade de conjuntos que ela possuirá, como pode ser observado na equação 3.2.

$$NúmeroConjuntos = \frac{NúmeroLinhas}{Associatividade} \quad (3.2)$$

Estes valores também são válidos nesta arquitetura, pois é proposta uma adaptativa para realizar a reconfiguração dinâmica da associatividade de memórias cache comuns, apenas com a implementação de algumas modificações estruturais e também com a adição de alguns elementos extras.

Esta proposta de arquitetura de memória cache reconfigurável tem como objetivo o superaproveitamento das linhas dentro do conjuntos da memória. Como visto anteriormente, as cargas de trabalho não costumam ter uma distribuição de acessos uniforme à memória. Por isso, é muito comum que alguns conjuntos dentro da cache sejam superutilizados, com mais acessos e faltas que outros como demonstra o exemplo da Figura 9.

Memória RAM															
0	1	2	3	6	1	A	B	7	8	E	F	0	3	2	6

Memória Cache			
0	8	Acessos = 3	Faltas = 2
1	X	Acessos = 1	Faltas = 1
6	2	Acessos = 6	Faltas = 6
3	F	Acessos = 5	Faltas = 5

Figura 9 – Exemplo de acessos não uniformes à memória

Como pode ser analisado nesta figura (Figura 9), os conjuntos 0 e 1 não

tiveram muitos acessos, e a pouca quantidade de faltas foi devido a falta compulsória (primeiro acesso à linha), enquanto os conjuntos 2 e 3 apresentaram uma grande quantidade de acessos e faltas.

A idéia é que em casos como estes, os conjuntos superutilizados possam receber as linhas de conjuntos menos utilizados, para tentar compensar a não uniformidade de acessos à memória. Na Figura 10, observa-se uma memória cache reconfigurável do tipo 4-8 way¹⁶ com sua configuração inicial e após uma rodada de reconfiguração. Como pode ser observado, a apresentada proposta de memória cache dá a possibilidade de um conjunto doar uma ou duas linhas, assim como os conjuntos receptores tem a possibilidade de receber uma ou duas linhas. Uma explicação mais detalhada sobre esta característica será realizada posteriormente, o interessante neste momento é observar a característica de recepção/doação para entender a sua importância na utilização em memórias cache.

Memória cache antes da reconfiguração

Memória cache após a reconfiguração

Figura 10 – Exemplo de doação/recepção de linhas entre conjuntos

Como dito, será utilizada uma memória cache associativa por conjuntos de linhas. Em cada conjunto de linhas da cache, dois registradores serão adicionados, um deles para contar a quantidade de acessos ao conjunto e o outro para contar a

¹⁶ X-Y way é o formato escolhido para definir que a memória reconfigurável possui associatividade inicial X e associatividade máxima Y [48].

classificação dos conjuntos: a barreira central, a inferior e a superior. Dependendo da posição do conjunto quanto a essas barreiras, eles irão obter seus rótulos para classificação. Os possíveis rótulos são: 2P, 1P, 1G e 2G. Para encontrar o valor da barreira central das faltas, que divide os valores 2P e 1P dos valores 1G e 2G, basta fazer uma média de todos os registradores de faltas e dividir pelo número total de conjuntos. Para encontrar a barreira inferior, basta somar o valor dos registradores de falta dos conjuntos que possuem valor inferior à barreira central, e após isso, dividir pelo número de conjuntos que possuem menos faltas que a barreira central. A barreira superior funciona de maneira similar, somando o valor de todos os registradores dos conjuntos que possuem valor de faltas superior a barreira central, e dividir o resultado pelo número de conjuntos que possuem mais faltas que a barreira central. A Figura 12 ilustra os rótulos para classificação de acordo com as barreiras. O mesmo procedimento é efetuado para definir os rótulos para classificação com relação ao registrador de acessos.

Figura 12 – Rótulos para classificação

Os conjuntos de linhas com valores menores ou iguais a barreira inferior receberão a rotulação de 2P, os valores menores ou iguais a barreira central receberão a rotulação 1P, os valores menores ou iguais a barreira superior receberão a rotulação 1G e os outros valores, acima da barreira superior, receberão a rotulação 2G.

O exemplo da Tabela 3 demonstra a obtenção dos valores das barreiras central, inferior e superior na análise de uma suposta memória cache, e a rotulação de cada um dos conjuntos desta memória cache utilizando os possíveis rótulos de classificação de acordo com o valor das barreiras obtidos.

Tabela 3 – Exemplo de rotulação

Conjunto	Faltas	Acessos	Rótulos das Faltas	Rótulos dos Acessos
0	10	300	2P	1P
1	20	400	1P	1P
2	110	1200	1G	2G
3	140	200	2G	2P
4	10	100	2P	2P
5	20	900	1P	1G
6	110	100	1G	2P
7	20	600	1P	1G
Barreira Central	55	475		
Barreira Inferior	16	220		
Barreira Superior	120	900		

Definidas as barreiras, os conjuntos serão classificados pela rotulação de seus registradores de acordo com o número de faltas e de acessos, onde os conjuntos classificados com as prioridades de menor valor, possuem maior prioridade para receber linhas, e os conjuntos classificados com o maior valor possuem maior prioridade para doar linhas. As possíveis classificações dos conjuntos podem ser observadas na Tabela 4.

Tabela 4 – Classificação

Prioridade	Faltas/Acessos	Ponteiros	Descrição
1	2G/2G		Recebe duas linhas se possível
2	2G/1G		
3	1G/2G		
4	1G/1G		
5	2G/1P	R2	Recebe uma linha se possível
6	2G/2P		
7	1G/1P		
8	1G/2P		
9	1P/2G	R1	Permanece estável
10	1P/1G		
11	2P/2G		
12	2P/1G	D1	
13	1P/1P		Doa uma linha se possível
14	1P/2P	D2	
15	2P/1P		Doa duas linhas se possível
16	2P/2P		

As prioridades foram definidas assim seguindo o conceito que conjuntos com muitos acessos e muitas faltas necessitam de mais linhas e os conjuntos com poucas faltas e poucos acessos podem doar linhas extras.

A partir da classificação de todos os conjuntos contidos na memória cache, dois tipos distintos de conjuntos doadores e receptores serão criados, os receptores que recebem duas linhas, se possível, por reconfiguração e os receptores que recebem uma linha por reconfiguração, e o mesmo para os doadores. Em ambos os casos, internamente aos grupos o conceito de prioridade será seguido, tendo maior prioridade de doar os conjuntos com prioridade de maior valor, e maior prioridade de receber, os conjuntos com prioridade de menor valor. Estes tipos de conjuntos são definidos por quatro ponteiros, dois para os doadores (D1 e D2) e dois para os receptores (R1 e R2). Para a doação, conjuntos com valor de prioridade maior que D2 doam duas linhas se possível, e os valores maiores que D1, doam uma linha. Nos receptores, os conjuntos com prioridade menor que R2 recebem duas linhas se possível, e os menores que R1, recebem uma linha. Os conjuntos menores ou iguais a D1 e maiores ou iguais a R1, permanecem estáveis.

Algumas ressalvas precisam ser feitas quanto aos conjuntos: um conjunto doador que possuir apenas uma linha não fará a doação; um conjunto receptor que possuir o valor máximo de y linhas em seu conjunto ($x-y$ way) não receberá linha extra; conjuntos destinados a receber duas linhas extras, só poderão receber as duas linhas se ao recebê-las não ultrapassarem o valor de y , e se possuírem algum doador que possa doar duas linhas ao mesmo tempo; conjuntos doadores de duas linhas só irão doar duas linhas se possuírem pelo menos três linhas em seu conjunto, e se existir pelo menos um conjunto que possa receber as suas duas linhas ao mesmo tempo; em hipótese alguma haverá acúmulo de conjuntos doadores para doar para um mesmo conjunto ou acúmulo de receptores para receber de um mesmo conjunto, neste caso, o receptor ou o doador receberá apenas uma linha durante a execução do algoritmo, podendo receber ou doar outra linha apenas na próxima reconfiguração.

Após esta análise detalhada das principais características desta proposta de algoritmo de reconfiguração bem como seu total funcionamento, pode-se fazer um pequeno resumo com as principais alterações que esta proposta adaptativa possui quando comparada com a proposta de algoritmo de reconfiguração da memória cache definida por Carvalho e Martins, alterações estas que foram propostas com o

intuito de compensar as deficiências analisadas neste algoritmo. As principais alterações desta proposta comparada com a proposta de Carvalho e Martins são:

- A inclusão de um sistema de barreiras mais justo para a rotulação dos conjuntos de linhas da memória cache quanto suas faltas e acessos;
- Aliado ao sistema de rotulação mais elaborado, a proposta de um sistema de classificação compatível;
- A possibilidade de ocorrer doação/recepção duplas, tornando o sistema mais ágil para se recuperar quando ocorrer uma inversão do padrão doador-receptor nos conjuntos de linhas da memória cache;
- A abertura a possibilidade de adição de mais duas camadas de adaptação, uma nos ponteiros dos doadores/receptores e outra na obtenção das barreiras de rotulação para posterior classificação.

4 SIMULADOR DESENVOLVIDO

Neste capítulo será feita uma apresentação inicial dos ambientes de simulação utilizados durante estes estudos, e após esta apresentação, será realizada uma análise detalhada do simulador utilizado para a obtenção dos resultados desta proposta juntamente com o ambiente computacional e os *traces* utilizados como dados de entrada.

4.1 Estudo de simuladores e *traces*

Durante a fase de preparação para as simulações, foram analisados diversos métodos de simulação para serem utilizados. Inicialmente testes foram realizados utilizando o simulador SimpleScalar [11] que tem como característica a emulação de conjuntos de instruções reais como os das arquiteturas Alpha, PISA, ARM e x86. As simulações realizadas com o SimpleScalar são completas, abrangendo toda arquitetura analisada, como pipeline, *branch prediction* e até mesmo simulação da memória cache, entre outros módulos de um processador que podem ser simulados com este programa. Algumas modificações foram realizadas e testadas no módulo sim-cache de memórias cache deste simulador para análises de resultados.

Após a utilização de um emulador como ambiente de simulação, outro projeto foi escolhido para análise de simulações, o LEON [12]. Leon é um processador SPARC V8 escrito em VHDL, que em conjunto com simuladores VHDL (ModelSim [14] ou a alternativa *opensource*, GHDL [13]) podem executar programas escritos para processadores SPARC. A melhor característica deste tipo de simulação é a possibilidade de implementação em *hardware* utilizando alguma FPGA.

Por fim, decidiu-se pela utilização de *traces* reais, que são um histórico com todos os acessos à memória realizados durante um dado intervalo de tempo. A utilização de *traces* foi selecionada, pois os *traces* são obtidos com a execução de *benchmarks* em sistemas reais, que levam em conta a influência do sistema operacional no resultado final. Inicialmente foram realizados testes utilizando o simulador de *trace* Dinero IV [49] para uma melhor adaptação a este ambiente de simulação. Após alguns testes com este simulador, decidiu-se pela utilização de um simulador próprio, desenvolvido durante as atividades para a realização das simulações para este trabalho.

Tabela 5 – Benchmarks utilizados nas simulações

<i>Benchmark</i>	Linguagem	Tipo	Descrição
164.gzip	C	Int	Compressão
168.wupwise	Fortran 77	FP	Cromodinâmica física / quântica
171.swim	Fortran 77	FP	Modelamento de fluidos
172.mgrid	Fortran 77	FP	Campos de potencial 3D
173.applu	Fortran 77	FP	Equações diferenciais parciais parabólicas / elípticas
175.vpr	C	Int	Place and route em circuitos FPGA
176.gcc	C	Int	Compilador C
177.mesa	C	FP	Biblioteca gráfica 3D
178.galgel	Fortran 90	FP	Dinâmica computacional de fluídos
179.art	C	FP	Reconhecimento de imagens / redes neurais
181.mcf	C	Int	Otimização combinatorial
183.equake	C	FP	Simulação de propagação de ondas sísmicas
186.crafty	C	Int	Xadrez
187.facerec	Fortran 90	FP	Processamento de imagens: reconhecimento facial
188.amp	C	FP	Química computacional
189.lucas	Fortran 90	FP	Teoria de números/ teste de primalidade
191.fma3d	Fortran 90	FP	Simulação de acidentes (números finitos)
197.parser	C	Int	Processamento de texto
200.sixtrack	Fortran 77	FP	Física nuclear: projeto de aceleradores
252.eon	C++	Int	Visualização computacional
253.perlbmk	C	Int	Linguagem de programação PERL
254.gap	C	Int	Teoria de grupos, interpretador
255.vortex	C	Int	Banco de dados orientado a objetos
256.bzip2	C	Int	Compressão
300.twolf	C	Int	Simulador place and route
301.apsi	Fortran 77	FP	Meteorologia

As simulações foram realizadas a partir de *traces* da BYU Trace Distribution Center [17] obtidos com os *benchmarks* SPEC CPU2000 [50], utilizando tanto os *benchmarks* de inteiros quanto os de ponto flutuante. Estes *traces* variam desde operações de compressão usando gzip e bzip2, compilações com gcc usando diversas cargas de entrada, a simulações de propagação de ondas sísmicas e jogos de xadrez. Diversas configurações são encontradas para a utilização, podendo ser encontrados *traces* utilizando máquinas com processadores Pentium Pro, Pentium II e Pentium III, e com diferentes sistemas operacionais como Windows 2000, Windows NT ou Linux Red Hat. Os *traces* escolhidos para simulação foram obtidos em máquinas Pentium III 733MHz com Windows 2000. Estes *traces* são descritos na Tabela 5 acima, informando também o tipo de dados que eles utilizam e a linguagem de programação em que estes *benchmarks* foram escritos.

4.2 Simulador Bobsim

Para a obtenção de resultados utilizando diversos modelos de memória cache (tanto os convencionais como os reconfiguráveis), foi desenvolvido um simulador de acessos à memória cache, que foi nomeado provisoriamente de BobSim [51]. Este simulador utiliza como carga de trabalho *traces* de acessos à memória da BYU Trace Distribution Center, obtidos com a execução de *benchmarks* do SPEC CPU2000 em sistemas reais, levando em consideração a ação do sistema operacional.

O BobSim é um simulador que foi desenvolvido para operar apenas em modo texto, para permitir a execução de diversas instâncias em *background* em *clusters* de computadores com o intuito de obter resultados de simulações de várias organizações de memórias cache para análise posterior. A escolha de execução em *background* ocorreu pois cada simulação, utilizando os *traces* obtidos no BYU, podem demorar de 1 a 45 minutos, dependendo da organização escolhida e da máquina utilizada para a execução do simulador.

O simulador foi escrito em linguagem C para computadores utilizando sistemas operacionais da família GNU/Linux, mas com certas alterações na sua estrutura, pode ser facilmente portado para a família Windows. Foram desenvolvidas duas versões para este simulador com diversas alterações para este estudo.

4.2.1 Bobsim versão 1.0

A primeira versão do simulador Bobsim possui uma arquitetura mais simples e não modular, com uma menor possibilidade de opções de configurações de memórias cache e o modo de operação mais primitivo, sem a possibilidade de configurar trabalhos em *batch* com um arquivo de configuração.

A arquitetura de memória utilizada nesta versão do simulador é a Von-Neumann (ou Princeton), onde as memórias de dados e instruções são unificadas, e diversas configurações podem ser definidas no início da execução utilizando linhas de comando.

As possíveis configurações que podem ser definidas são:

- Algoritmo de substituição – LRU, MRU, FIFO ou LIFO;

- Algoritmo de associatividade – os tipos de associatividade implementados são: as organizações tradicionais (mapeamento direto, associativo por conjuntos e totalmente associativo), a organização com associatividade reconfigurável Carvalho e Martins [48] e a organização com associatividade reconfigurável Kerr e Midorikawa [18];
- Associatividade inicial dos conjuntos – valores na potência de 2 para a associatividade inicial dos conjuntos (quando utilizado uma organização tradicional, este é o valor da associatividade da memória cache);
- Associatividade final dos conjuntos – valores na potência de 2 para a associatividade final dos conjuntos (quando utilizada a organização tradicional, esta configuração não é utilizada);
- Tamanho da memória cache – Tamanho em KB da memória cache com valores na potência de 2;
- Tamanho da linha – Tamanho em *bytes* das linhas da memória cache com valores na potência de 2 e tamanho mínimo de 8 *bytes*.

Por ser desenvolvido como um simulador para ser executado em *background*, a melhor forma de se apresentar os resultados é na criação de arquivos de saída. Foram definidos dois tipos de arquivos de saída que sempre são gerados juntamente após as execuções, todos são utilizados na forma de *update* (se o arquivo não existir é gerado, e se existir, os dados novos são incluídos no final do arquivo). As possíveis formas de gerar os arquivos são:

- Log de estatísticas básicas – Esta forma gera um arquivo de texto (*.log) com informações gerais sobre as simulações, como as configurações iniciais, o número de referências totais à memória, o número de acessos a memórias de instruções (independente da arquitetura escolhida), o número de leituras e escritas na memória de dados, o número de faltas compulsórias, o número total de faltas e a taxa de faltas;
- Dados para planilhas – Esta forma gera um arquivo de texto (*.plan) apenas com as configurações da memória cache e com a taxa de falta, de forma adequada (apenas os valores, sem legendas) para fácil adição em planilhas de texto. Este formato de arquivo é interessante para a análise gráfica comparativa de grandes quantidades de simulações;

$\{\text{asso}\}-\{\text{subs}\}-\{\text{trace}\}. \{*\}$

Figura 13 – Formato dos arquivos de log

A estrutura do nome dos arquivos gerados com as simulações realizadas nesta versão do simulador é definida na Figura 13, onde:

- Asso – é o algoritmo de associatividade utilizado;
- Subs – é o algoritmo de substituição utilizado;
- Trace – é o *trace* utilizado;
- * – é o tipo do arquivo gerado.

4.2.2 Bobsim versão 2.0

A necessidade de um simulador com código modular que facilitasse a modificação de seu funcionamento e possibilitasse uma inclusão mais fácil de novos algoritmos, acabou levando ao desenvolvimento de uma nova versão do simulador Bobsim, totalmente reestruturada e com a adição de novas funcionalidades.

Nesta versão atual, as simulações podem ser realizadas nos níveis 1,2 e 3 e a arquitetura da memória cache utilizada pode ser tanto a unificada (Princeton) quanto a não-unificada (Harvard), onde os barramentos da memória de dados e da memória de instruções são separados.

Diversas configurações podem ser definidas antes da execução da simulação. As possíveis configurações que podem ser definidas são:

- Algoritmo de substituição – FIFO, LIFO, LRU e MRU;
- Tipo de associatividade – os tipos de associatividade implementados são: as organizações tradicionais (mapeamento direto, associativo por conjuntos e totalmente associativo), a organização com associatividade reconfigurável Carvalho e Martins e a organização com associatividade reconfigurável Kerr e Midorikawa;
- Associatividade inicial dos conjuntos – valores na potência de 2 para a associatividade inicial dos conjuntos (quando utilizado uma organização tradicional, este é o valor da associatividade da memória cache);

- Associatividade final dos conjuntos – valores na potência de 2 para a associatividade final dos conjuntos (quando utilizada a organização tradicional, esta configuração não é utilizada);
- Tamanho da memória cache – Tamanho em KB da memória cache com valores na potência de 2;
- Tamanho da linha – Tamanho em *bytes* das linhas da memória cache com valores na potência de 2 e tamanho mínimo de 8 *bytes*;
- Quantidade de *levels* – Quantidade de *levels* (ou níveis) utilizadas na simulação, 1, 2 ou 3;
- Tipo de arquitetura – Tipo de arquitetura utilizada nas simulações, Harvard ou Von-Neumann/Princeton.

Na primeira versão do BobSim, os dados de configuração inicial eram utilizados para a configuração da memória integrada em L1, mas com as alterações da versão nova, cada um dos níveis podem ser configuradas diferentemente, e, em caso de memórias separadas, a memória de dados e instruções também podem ser configuradas de forma independente.

Alguns campos das configurações ainda possuem poucas opções para ser utilizadas, porém, graças à estrutura modular do código do simulador, a implementação de novas opções para serem utilizadas nas simulações é bastante simples. A ideia é adicionar o máximo de opções possíveis para simulações de caches tradicionais ou dos diversos algoritmos de caches reconfiguráveis, tornando esta uma ferramenta ideal de simulação e aprendizado de memórias cache.

Assim como a primeira versão deste simulador, esta versão também é executada em *background* no sistema, sendo a melhor forma de se apresentar os resultados é na criação de arquivos de saída. Definimos três tipos de arquivos de registros de resultados das simulações foram definidos que podem ser utilizados em qualquer combinação possível, todos são utilizados na forma de *update*. As possíveis formas de gerar os arquivos são:

- Log de estatísticas básicas – Esta forma gera um arquivo de texto (*.log) com informações gerais sobre as simulações, como as configurações iniciais, o número de referências totais à memória, o número de acessos a memórias de instruções (independente da arquitetura escolhida), o número de leituras e

escritas na memória de dados, o número de faltas compulsórias, o número total de faltas e a taxa de faltas;

- Dados para planilhas – Esta forma gera um arquivo de texto (*.plan) apenas com as configurações da memória cache e com a taxa de falta, de forma adequada (apenas os valores, sem legendas) para fácil adição em planilhas de texto. Este formato de arquivo é interessante para a análise gráfica comparativa de grandes quantidades de simulações;
- Log de informações detalhadas – Esta forma gera um arquivo de texto (*.full) com informações completas sobre as simulações, como a informação detalhada sobre as faltas de cada uma das linhas da memória, informações sobre a ocorrência de doações de linhas (no caso de algoritmos reconfiguráveis) entre outras informações para uma análise mais detalhada sobre o comportamento da organização escolhida desde o início até o final da execução de sua simulação.

1 - {nível}-{unif}-{asso}-{subs}-{trace}.{*}

2 - {nível}-{unif}-{asso}-{subs}-{trace}-{size}_{cols}_{max}_{line}.full

Figura 14 – Formato dos arquivos de log

A estrutura do nome dos arquivos gerados com as simulações realizadas é definida na Figura 14, onde os arquivos (1) são gerados para os log *.log e *.plan e o (2) é o formato dos arquivos gerados para os *.full, sendo:

- Nível – é o nível da memória que o *log* pertence;
- Unif – é o tipo de cache, unif, data ou inst;
- Asso – é o algoritmo de associatividade utilizado;
- Subs – é o algoritmo de substituição utilizado;
- Trace – é o *trace* utilizado;
- Size – é o tamanho da memória cache;
- Cols – é o tamanho inicial da associatividade;
- Max – é o tamanho máximo da associatividade;
- Line – é o tamanho da linha;
- * – é o tipo do arquivo gerado.

Na primeira versão do BobSim, o simulador era um programa monoprocessado, possuindo apenas uma *thread* que realizava as simulações, precisando ser executado várias vezes caso fosse desejado uma grande quantidade de simulações para comparação de resultados. Esta lista foi facilmente gerada em um arquivo de *batch* que ia executando as simulações à medida que elas eram necessárias. Uma grande modificação na segunda versão foi a utilização da biblioteca MPI no simulador para um melhor desempenho deste em um cluster computacional. O simulador utiliza a estratégia *self-scheduling*, como mostra a Figura 15.

Figura 15 – Modelo self-scheduling

Com a utilização deste esquema, o simulador funciona com um programa mestre (M) que é o responsável pelo escalonamento de tarefas para serem executadas pelos programas escravos (E), inclusive para si mesmo. Com esta grande mudança, o simulador ficou mais eficaz para a realização de grandes quantidades de simulações para uma análise posterior, algo muito interessante no estudo de arquiteturas de computadores, para um melhor entendimento das diversas organizações de memórias caches propostas e existentes.

Uma outra característica interessante adicionada nesta versão do BobSim para a simulação em sistemas paralelos é a possibilidade da criação de um arquivo de configuração xml com o nome dos *traces* a serem simulados, juntamente com as possíveis configurações a ser utilizadas, possibilitando também a utilização de intervalos de configuração com qualquer um dos campos numéricos da configuração. Esta opção possibilita, por exemplo, fazer que o mestre possa escalonar serviços numa certa configuração com associatividade inicial 1

(mapeamento direto) e ir aumentando este valor até 8 (mapeamento associativo por conjuntos), possibilitando a comparação de memórias cache com a mesma carga de trabalho e as mesmas características e tamanho, porém com associatividades diferentes. Um pequeno trecho do arquivo xml pode ser observado a seguir:

```
<Layer1>
  <Size>
    <Init>32</Init>
    <End>2048</End>
  </Size>
  <LineSize>
    <Init>16</Init>
    <End>128</End>
  </LineSize>
  <NumCols>
    <Init>1</Init>
    <End>32</End>
  </NumCols>
  <MaxNumCols>
    <Init>2</Init>
    <End>64</End>
  </MaxNumCols>
  <Subst>LRU</Subst>
  <Assoc>KERR</Assoc>
  <Unified>YES</Unified>
</Layer1>
```

Este arquivo de configuração não é utilizado apenas para simulações em *clusters*, mas pode ser utilizado em *workstations* para facilitar a simulação, substituindo os arquivos *batch*.

4.3 Ambiente computacional

O ambiente computacional utilizado para realizar as simulações foi um *cluster* com dez máquinas, todas utilizando distribuições Linux. Dentre as máquinas utilizadas para simulação, duas máquinas (Intel1 e Intel2) são equipadas com processadores Pentium D 950 3.40GHz, com 2GB de memória RAM, e as outras oito máquinas utilizadas (Bio01 a Bio08) são equipadas com dois processadores AMD Athlon MP 2400+, com 1GB de memória RAM.

5 MEMÓRIAS CACHE UNIFICADAS

Neste capítulo será analisado o desempenho dos algoritmos de associatividade em memórias cache unificadas. Para isso, serão apresentadas três análises interessantes obtidas através dos resultados das simulações: O padrão receptor-doador, a prioridade diferenciada na recepção e doação, e o caso especial da configuração 4-8 way. Ao final deste capítulo, serão relatadas as conclusões sobre os resultados já obtidos após as simulações realizadas.

5.1 Simulações Realizadas

Atualmente, diversas configurações de memória cache podem ser encontradas, variando tanto seu tamanho total, como sua associatividade, tamanho de linha, algoritmo de substituição e até mesmo a arquitetura, mesmo que raramente. Para as simulações a seguir, será utilizado cache de tamanho variando entre 32KB a 256KB, associatividade inicial variando de 2 a 16 e associatividade final variando de 4 a 32. O tamanho da linha escolhido foi de 32 *bytes*, por ser um valor bastante comum.

Diversas análises comparativas foram feitas em alguns modelos de memória cache (tradicionais e reconfiguráveis) para se obter uma análise de desempenho. Inicialmente, simulações foram realizadas utilizando modelos de cache tradicionais com mapeamento direto, associativo por conjuntos e totalmente associativo para fins de comparação de desempenho. A seguir foi analisado o algoritmo Carvalho e Martins [48], que serviu como parâmetro de comparação de algoritmos reconfiguráveis. Um exemplo destas simulações pode ser observado na Figura 16. Essa figura apresenta uma comparação dos desempenhos das organizações tradicionais de memória cache (mapeamento direto, associativo por conjuntos e totalmente associativo) com a configuração 4-8 way do algoritmo Carvalho e Martins para o *trace* 164 gzip graphic.

Figura 16 – Desempenho de diferentes caches

Os resultados obtidos mostram que na simulação com esta carga de trabalho o algoritmo Carvalho e Martins apresenta um desempenho melhor que as organizações mapeamento direto e associativo por conjuntos, contudo, com um desempenho ainda pior que a organização totalmente associativa. Isto mostra que ainda é possível obter melhorias de desempenho.

5.1.1 Caso 1: Padrão receptor-doador

Apesar dos resultados obtidos com as simulações da Figura 16, na sua grande maioria, terem sido como esperado (resultados obtidos com a memória cache totalmente associativa com o melhor desempenho entre as organizações, seguido dos resultados obtidos através da memória cache utilizando o algoritmo Carvalho e Martins e por fim os resultados obtidos utilizando as memórias cache com organização associativa por conjuntos¹⁸ e as memórias cache com mapeamento direto), em alguns casos o algoritmo Carvalho e Martins obteve resultados insatisfatórios, com a taxa de faltas superior às taxas obtidas utilizando a AC.

As piores se deram devido ao fato que os *traces* não seguem um padrão receptor-doador estável, sendo que consideramos como um padrão receptor-doador estável quando os conjuntos mais acessados e com mais faltas estão sempre

¹⁸ Como o termo organização associativa por conjuntos é muito usado neste trabalho, iremos adotar a sigla “AC” ou os termos “arquitetura normal” e “algoritmo normal” para designar esta organização de cache.

dispostos a receber linhas e sempre serão considerados conjuntos receptores, e os conjuntos com as menores quantidades de acessos e faltas estão sempre dispostos a doar linhas e, assim como os conjuntos receptores, sempre serão considerados conjunto doadores.

252.eon kajiya, size=64KB, association 8-16

Figura 17 – Análise da influência da inversão no padrão doador-receptor

Na Figura 17 é possível observar um pequeno trecho de momentos similares do *log* completo de doações e recepções dos algoritmos adaptativos de associatividade. Cada linha da imagem representa as doações (vermelho) e as recepções (azul) ocorridas em uma execução do algoritmo adaptativo, e cada coluna representa um dos conjuntos da memória cache. Como pode ser observado, num mesmo intervalo de tempo total analisado, o algoritmo Carvalho e Martins (Figura

17.a) realizou muito mais doações e recepções de um mesmo conjunto que o algoritmo Kerr e Midorikawa (Figura 17.b), cuja causa é a demora que o algoritmo Carvalho e Martins tem para se readaptar a uma inversão no padrão doador-receptor.

Como analisado previamente, os acessos às memórias não são uniformes, e uma uniformidade aparente dos acessos à memória em um sistema como este faria com que a memória se adaptasse a esta possível uniformidade, porém, com uma inversão no padrão de acesso, este algoritmo iria ter que perder algumas rodadas de adaptação para ficar num novo estado teoricamente estável. Mas alguns *benchmarks*, ou mesmo cargas de trabalho, realizam esta inversão no padrão de acessos constantemente.

Dado este fato, este algoritmo (Carvalho e Martins) se mostrou frágil, pois conjuntos previamente definidos como conjuntos receptores ou conjuntos doadores levam mais tempo para se readaptarem para um novo padrão de acessos onde os mais prováveis a receber linhas se tornam mais prováveis a doar e vice-versa.

A presente proposta de algoritmo busca resolver este tipo de problema permitindo a doação ou recepção de até duas linhas por reconfiguração. Como pode ser observado na Figura 18, o algoritmo proposto conseguiu melhoras de desempenho sobre o algoritmo Carvalho e Martins e sobre a AC em diferentes casos.

Figura 18 – Influência da alteração no padrão doador-receptor nas simulações realizadas

A Figura 18.a apresenta os resultados obtidos para o *trace* 252.eon kajiya. De uma maneira geral o algoritmo Carvalho e Martins apresenta resultados piores que a AC. Por exemplo, para o tamanho de cache de 32KB a taxa de falta é 0,3% superior a AC. Nesta configuração o algoritmo proposto apresentou resultados melhores que ambas as organizações com uma taxa de faltas de 11,9%, correspondendo a uma melhora de 0,81% em relação ao algoritmo Carvalho e Martins. Um ganho maior foi obtido para um cache de 64KB com uma taxa de faltas 1,29% menor que a taxa do algoritmo Carvalho e Martins.

Resultados semelhantes foram obtidos com o *trace* 253.perlbnk perfect. O algoritmo proposto apresentou taxas de faltas menores de cerca de 1% que o algoritmo Carvalho e Martins para ambos os tamanhos de cache.

Esta comparação mostra que o algoritmo aqui proposto se adapta mais facilmente às mudanças no padrão receptor-doador das aplicações, que é uma das deficiências do algoritmo Carvalho e Martins. Esta melhor adaptação do algoritmo proposto a situações de inversão total do padrão receptor-doador é dada pela possibilidade da doação e recepção de até duas linhas por vez.

5.1.2 Caso 2: Prioridade diferenciada na recepção e doação

Uma das idéias iniciais com a proposta deste algoritmo era a possibilidade de realizar uma política de priorização diferenciada em relação aos possíveis doadores e receptores. Com a inclusão de um sistema de barreiras mais desenvolvidos e, conseqüentemente, com o aumento de possíveis rótulos de classificação foi possível obter um sistema que fosse mais justo neste requisito de projeto.

Nas situações realizadas, como dito anteriormente, houve casos em que o algoritmo Carvalho e Martins obteve os resultados esperados, ou seja, com desempenho superior aos resultados obtidos com memórias cache utilizando associatividade por conjunto, porém inferior ainda aos resultados obtidos em memórias cache totalmente associativas, que possibilitam uma melhoria de desempenho ainda maior do que a já obtida pelo algoritmo Carvalho e Martins. Nestas simulações, a presente proposta busca obter melhoras adicionais de desempenho com a sua definição de um número maior de classificação de tipos de conjuntos. Isto permite diferenciar melhor os vários conjuntos de acordo com o número de acessos e faltas atribuindo um número maior de prioridades. Na Figura

19, é possível ver alguns exemplos de casos onde o algoritmo Carvalho e Martins possui um bom desempenho sobre a AC, e uma comparação com a proposta aqui apresentada.

Figura 19 – Influência da prioridade diferenciada na doação e recepção de linhas nas simulações realizadas

Na Figura 19.a, com resultados obtidos com o *trace* do *benchmark* 255.vortex two, para tamanhos de cache de 32KB e 64KB, o algoritmo Carvalho e Martins tem

um ganho no desempenho, diminuindo a taxa de falta em aproximadamente 0,15% e 0,35%, comparado com o valor obtido com as memórias associativas por conjunto. Os resultados obtidos para o algoritmo aqui proposto mostra uma diminuição da taxa de faltas em relação ao algoritmo Carvalho e Martins em cerca de 0,95% em ambos os tamanhos. Resultados semelhantes foram obtidos para os *traces* obtidos com os *benchmarks* 176.gcc expr (Figura 19.b), 177.mesa (Figura 19.c) e 254.gap (Figura 19.d) onde as memórias cache utilizando o algoritmo proposto apresentaram uma taxa de faltas menor que as memórias cache utilizando o algoritmo Carvalho e Martins em 0,8%, 0,85% e 0,2% respectivamente.

Estes resultados comprovam que algoritmo proposto atingiu as nossas metas pré-estipuladas, conseguindo obter uma diferenciação superior para os conjuntos com uma maior necessidade de receber linhas pois acaba atribuindo uma prioridade maior em relação aos outros conjuntos, graças à proposta de uma melhor classificação dos conjuntos, fato que foi obtido com o aumento de barreiras de classificação.

5.1.3 Caso 3: Configuração 4-8 way

A partir das inúmeras simulações realizadas, foi possível observar que a presente proposta de algoritmo em algumas configurações iniciais acabou se sobressaindo sobre outras arquiteturas como o algoritmo Carvalho e Martins e as memórias cache utilizando associatividade por conjunto, quando se comparado o ganho de desempenho pela taxa de faltas obtida em cada uma dessas simulações.

Com base nos resultados obtidos pelas simulações realizadas, o próximo estudo realizado acabou por verificar a existência de uma configuração de cache reconfigurável que sempre apresentasse ganhos de desempenho em todas as aplicações utilizadas nas simulações realizadas. Desta forma esta configuração poderia ser utilizada como uma configuração padrão para as memórias cache reconfiguráveis em sistemas reais.

As simulações realizadas apontaram para uma configuração especial com a política proposta de adaptação: caches utilizando a configuração 4-8 way obtiveram, em todas simulações, excelentes resultados comparados ao algoritmo Carvalho e Martins e ao associativo por conjuntos. A Figura 20 mostra alguns resultados obtidos por esta configuração.

Figura 20 – Configuração 4-8 way

Na Figura 20.a, o *trace* de *benchmark* com ponto flutuante 171.swim é utilizado, apresentando uma taxa de faltas de 13,7% no associativo por conjuntos com cache de 32KB, uma taxa de 12,4% no algoritmo Carvalho e Martins e de 12% na proposta aqui apresentada. Os resultados seguiram esse padrão de queda para caches de 64KB, 128KB e 256KB, com uma taxa de faltas de 11,9%, 10,6% e 9,5% no associativo por conjuntos, e caindo aproximadamente 1.5% em todos os tamanhos para o algoritmo Carvalho e Martins, e o algoritmo aqui apresentado,

caindo mais 0,5% para 64KB, e 0,3% para 128KB e 256KB. As Figura 20.b a Figura 20.d mostram melhoras de cerca de 0,3% a 0,8% nos *traces* 186.crafty, 200.sixtrack e 175.vpr place. Resultados semelhantes foram obtidos com os outros *traces* analisados.

Concluiu-se assim que as memórias cache utilizando o algoritmo apresentado neste trabalho possuem uma configuração inicial preferencial, a 4-8 way, já que foi possível comprovar que em todos os casos simulados foram obtidos ganhos de desempenho. Esta configuração possui essa superioridade em relação a outras configurações pois consegue se aproveitar diretamente do caso anterior, pois inicialmente, esta configuração possui 3 linhas por conjuntos para doação, assim, esta configuração pode se adaptar mais rápido aos requisitos iniciais das simulações. Outra análise interessante é que esta é uma configuração bastante comum, muitos processadores atuais utilizam tanto associatividade por conjuntos inicial 4-way quanto a 8-way.

5.2 Conclusões dos resultados obtidos

A partir de resultados obtidos em nas simulações aqui apresentadas foi possível concluir que a presente proposta consegue obter melhorias no desempenho em relação às organizações de caches tradicionais e ao algoritmo Carvalho e Martins.

Um caso inicial foi analisado onde o algoritmo Carvalho e Martins não conseguia obter um desempenho melhor que o associativo por conjuntos, e a proposta aqui apresentada obteve melhores desempenhos devido ao fato que nosso algoritmo possui uma característica de melhor recuperação quando ocorre uma inversão no padrão doador-receptor dos conjuntos da memória cache.

Outro caso analisado foi a melhoria do desempenho mesmo nos casos em que o algoritmo Carvalho e Martins já possuía bons resultados comparado ao associativo por conjuntos. Essa melhoria ocorreu devido ao fato que a proposta feita neste trabalho possui uma política de prioridades mais detalhada para classificação de doadores e receptores.

O terceiro estudo realizado mostrou que existe uma configuração de cache reconfigurável que sempre apresenta ganhos de desempenho, a configuração 4-8 way. Esse resultado é muito importante, pois pode ser utilizado como configuração padrão de caches reconfiguráveis.

6 MEMÓRIAS CACHE NÃO UNIFICADAS

Neste capítulo será analisado o desempenho dos algoritmos de associatividade em memórias cache não unificadas (memórias cache de dados e instruções separadas). Também será apresentada a proposta de uma configuração ideal para arquiteturas de memórias cache não unificadas.

6.1 Simulações realizadas

Dada a situação atual das arquiteturas de processadores, onde a grande parte dos processadores está utilizando a arquitetura Harvard, é necessária uma análise comparativa entre as possíveis organizações de memórias cache com as memórias de dados e instruções separadas.

Para esta análise, decidiu-se gerar resultados com as características de memórias cache L1 mais utilizadas, para uma comparação entre as possíveis organizações. Para o tamanho das memórias cache de dados e instruções, foram utilizadas memórias de 16KB e 32KB tanto para dados como para instruções (com um total de 32KB e 64KB respectivamente), que são valores bastante utilizados para caches L1. As associatividades simuladas foram as 2-way, 4-way e 8-way, e para os algoritmos Carvalho e Martins e Kerr e Midorikawa, foram utilizadas as associatividades 2-4 way, 4-8 way e 8-16 way, utilizando memórias que possuem inicialmente a associatividade definida, e possuam no máximo o dobro de comparadores. Todas as simulações foram realizadas com memórias cache com linhas de 32 *bytes*, que é o valor mais utilizado entre as memórias caches e os *traces* utilizados foram os mesmos da Tabela 5.

6.1.1 Resultados obtidos

Durante as simulações, foi possível observar que algoritmo Carvalho e Martins possuía um desempenho superior ao algoritmo Kerr e Midorikawa para as memórias cache de instruções (Figura 21.a), e para as memórias cache de dados, o algoritmo Kerr e Midorikawa possuía um desempenho superior ao algoritmo Carvalho e Martins (Figura 21.b).

(a)

(b)

Figura 21 – Cache de instruções e de dados

Este comportamento deve-se ao fato que o algoritmo Carvalho e Martins demora mais tempo para se adaptar que o algoritmo Kerr e Midorikawa e os acessos à memória de instruções são mais uniformes que os acessos à memória de dados, o que torna um algoritmo que realiza menos mudanças em suas características mais eficiente que um algoritmo que realiza muitas mudanças.

Com essa análise sobre o comportamento dos dois algoritmos, propusemos a criação de uma memória cache híbrida, utilizando o algoritmo Carvalho e Martins para a memória cache de instruções e o algoritmo Kerr e Midorikawa para a memória cache de dados. Os resultados obtidos fazem uma comparação entre quatro tipos de memória cache, a memória sem adaptatividade na associação, a memória utilizando apenas algoritmo Carvalho e Martins, a memória utilizando apenas algoritmo Kerr e Midorikawa e a memória híbrida, utilizando o algoritmo Carvalho e Martins para a cache de instruções e o algoritmo Kerr e Midorikawa para a cache de dados.

Os resultados foram divididos em quatro categorias, sendo as três primeiras categorias para a divulgação dos resultados obtidos com *traces* de *benchmarks* científicos de programas mais conhecidos, como a utilização do programa de compactação gzip com diversas cargas de entradas, o compilador gcc com diversas cargas de entradas e o programa de compactação bzip2 também sendo utilizado

com diversas cargas de entrada. A quarta categoria envolve alguns resultados de outros *traces*, com resultados bastante interessantes quando comparados com outras arquiteturas.

Dentro das quatro categorias definidas, foram criadas duas sub-categorias, uma sub-categoria para os resultados das memórias de 32 KB e outra sub-categoria para os resultados das memórias de 64 KB. As categorias foram divididas deste modo para uma melhor análise individual das memórias cache.

6.1.1.1 Simulações com *traces* do gzip

A primeira análise sobre os resultados obtidos com a memória cache não unificada é a análise dos *traces* do gzip, que realiza tarefas de compressão de diversos tipos de dados. Os resultados obtidos com este *trace* em memórias cache unificadas não eram muito satisfatórios, sendo que em alguns casos o algoritmo Kerr e Midorikawa possuía um desempenho bastante inferior ao Carvalho e Martins. Com esta nova proposta foi possível obter resultados bastantes interessantes, sempre com melhorias, comparado com os outros algoritmos.

(a)

(b)

(c) (d)
Figura 22 – Resultados com cache de 32KB não unificadas com gzip

A Figura 22 demonstra alguns dos resultados obtidos com alguns *traces* do gzip para memórias cache de 32 KB. A Figura 22.a representa os resultados obtidos com o *trace* 164.gzip log com associatividade 2-4 way. A taxa de faltas total para a memória cache utilizando uma arquitetura não reconfigurável foi de 8,87%, e para o algoritmo Carvalho e Martins foi de 7,13%, um pouco melhor que os resultados obtidos para o algoritmo Kerr e Midorikawa, que obteve 7,27%. Já o algoritmo híbrido obteve uma pequena melhora com a taxa de faltas total de 7,05%, uma melhoria total de 1,17%¹⁹.

Para as outras associatividades, os resultados foram bem significativos. Com a associatividade 4-8 way com o *trace* 164.gzip graphic (Figura 22.b), o algoritmo híbrido conseguiu uma melhoria total de 3,77% sobre o algoritmo Carvalho e Martins, que por sua vez conseguiu uma melhoria de 3,53% sobre o algoritmo Kerr e Midorikawa e uma melhoria de 21,14% sobre o algoritmo normal. Com a mesma associatividade, 4-8 way, e com o *trace* 164.gzip random (Figura 22.d), obteve-se um dos melhores resultados dos *traces* de gzip com 32 KB, com o algoritmo híbrido obtendo um ganho de 3,56% sobre o algoritmo Carvalho e Martins. Por fim, utilizando a associatividade 8-16 way e o *trace* 164.gzip source (Figura 22.c), o algoritmo híbrido conseguiu diminuir a taxa de faltas em 2,75% em relação ao segundo melhor resultado desta mesma simulação, o algoritmo Carvalho e Martins.

¹⁹

Ganho percentual de um algoritmo sobre outro é calculado da forma: $(Miss_A - Miss_B) / Miss_A * 100\%$

Figura 23 – Resultados com cache de 64KB não unificadas com gzip

A Figura 23 representa os resultados obtidos com o *trace* gzip para memórias cache com um total de 64 KB. A Figura 23.a demonstra os resultados obtidos com o *trace* 164.gzip random para cache com 2-4 way. Como pode ser observado, o algoritmo Carvalho e Martins obteve um desempenho muito superior ao algoritmo normal, com um ganho total de 24,40%. O algoritmo Kerr e Midorikawa por sua vez teve o desempenho um pouco inferior ao Carvalho e Martins, obtendo um aumento na taxa de faltas de 0,5%. Já o algoritmo híbrido conseguiu melhorar o desempenho, e obteve uma taxa de faltas 1,5% menor que o algoritmo Carvalho e Martins. Para as outras simulações, utilizando o *trace* 164.gzip program com 4-8 way (Figura 23.b), 164.gzip graphic também com 4-8 way (Figura 23.d) e 164.gzip source com 8-16 way (Figura 23.c) a melhoria do algoritmo híbrido foi mais acentuada,

respectivamente, em 5,42%, 5,33% e 3,86% em relação ao algoritmo Carvalho e Martins, que obteve o segundo melhor desempenho nas respectivas simulações.

6.1.1.2 Simulações com *traces* do gcc

Nesta segunda análise, foram realizadas simulações com *traces* de gcc compilando com diversos arquivos de entrada. Estes *traces* também são muito interessantes por representarem resultados de ações que podem ser realizadas normalmente.

Figura 24 – Resultados com cache de 32KB não unificadas com gcc

Na Figura 24 é possível observar alguns dos resultados obtidos com os *traces* do gcc utilizando memórias de 32 KB. Os resultados para os resultados da Figura 24.a (176.gcc integ com 2-4 way) e Figura 24.b (176.gcc 200 com 4-8 way) são bem

similares. O algoritmo normal conseguiu taxa de faltas de, respectivamente, 15,30% e 13,81%. O algoritmo Carvalho e Martins conseguiu uma melhoria na taxa de faltas de 9,55% e 5,41% sobre os valores do algoritmo normal, e o algoritmo Kerr e Midorikawa conseguiu uma melhoria de 0,44% e 1,23% em relação ao algoritmo Carvalho e Martins. O algoritmo híbrido, por sua vez, conseguiu um ganho no desempenho de 0,53% e 1,45% em relação ao algoritmo Kerr e Midorikawa. Os resultados obtidos com os *traces* 176.gcc expr (Figura 24.c) e 176.gcc 166 (Figura 24.d) com associatividade 8-16 way para os algoritmos testados foram similares. O ganho do algoritmo híbrido sobre o segundo melhor algoritmo, o algoritmo normal, foi de, respectivamente, 2,25% e 0,93%.

(a)

(b)

(c)

(d)

Figura 25 – Resultados com cache de 64KB não unificadas com gcc

Utilizando memórias cache de 64 KB foi possível obter resultados mais interessantes com os *traces* do gcc. A Figura 25.a (176.gcc integ com associatividade 2-4 way), Figura 25.b (176.gcc expr com associatividade 4-8 way), Figura 25.c (176.gcc scilab com associatividade 8-16 way) e Figura 25.d (176.gcc 200 com associatividade 8-16 way) apresentam resultados significantes para o algoritmo Kerr e Midorikawa, com uma taxa de faltas de 10,87%, 10,58%, 10,11% e 9,54% respectivamente, sendo este o segundo melhor resultado obtido, ficando apenas atrás do algoritmo híbrido, que obteve melhoras de 1,04%, 1,83%, 3,61% e 4% sobre os resultados obtidos sobre o algoritmo Kerr e Midorikawa e de 18,25%, 13,77%, 8,9% e 6,54% sobre os resultados obtidos com a arquitetura com o algoritmo normal.

6.1.1.3 Simulações com *traces* do bzip2

O terceiro conjunto de simulações foi com outro *trace* de grande importância por ser baseado em um programa que é muito utilizado no dia a dia. O bzip2 é um compactador muito popular em sistemas operacionais GNU/Linux. Este compactador utiliza como dado de entrada um arquivo TIFF bem grande (gx), um binário de programa (px) e um código fonte (sx), onde cada arquivo é comprimido e descomprimido com dois diferentes graus de compactação, simulando uma ação normal de compactação do dia a dia.

(a)

(b)

(c)

(d)

Figura 26 – Resultados com cache de 32KB não unificadas com bzip2

Para todos os resultados obtidos com memórias cache de 32 KB, o algoritmo Carvalho e Martins obteve um resultado com desempenho total superior aos resultados obtidos com os algoritmos normal e Kerr e Midorikawa pelo fato que o seu desempenho com as memórias cache de instruções foi muito superior aos outros algoritmos, consequentemente o algoritmo híbrido obteve resultados melhores que o Carvalho e Martins, com ganhos de 1,88% para o *trace* 256.bzip2 p7 com associatividade 2-4 way (Figura 26.a), 3,49% para o *trace* 256.bzip2 s7 com 4-8 way (Figura 26.b), 4,26% para o *trace* 256.bzip2 g9 com associatividade 8-16 way (Figura 26.c) e 3,8% para o *trace* 256.bzip2 g7 utilizando a 4-8 way (Figura 26.d).

(a)

(b)

(c) (d)
Figura 27 – Resultados com cache de 64KB não unificadas com bzip2

Assim como os resultados para memórias de 32 KB, as memórias de 64 KB apresentaram resultados onde o algoritmo Carvalho e Martins foi superior aos outros algoritmos, com taxas de faltas de 6,34181% para o *trace* 256.bzip2 p7 com 2-4 way (Figura 27.a), 6,20811% para o 256.bzip2 g9 com 4-8 way (Figura 27.b), 5,29248% para o 256.bzip2 s9 com 8-16 way (Figura 27.c) e 4,95541% para o 256.bzip2 g9 com 8-16 way (Figura 27.d). O algoritmo híbrido, mais uma vez obteve um desempenho superior aos outros algoritmos impulsionado com o bom desempenho do algoritmo Carvalho e Martins para a cache de instruções e o algoritmo Kerr e Midorikawa para a cache de dados, obtendo ganhos de, respectivamente, 1,26%, 3,71%, 4,03% e 5,34% sobre o algoritmo Carvalho e Martins.

6.1.1.4 Outras simulações

Para as simulações apresentadas até agora, pudemos observar a capacidade de melhoria no desempenho total do algoritmo híbrido (algoritmo Carvalho e Martins para instruções e Kerr e Midorikawa para dados) com *traces* baseados em programas muito executados diariamente por diversos usuários.

Mas para uma melhor análise sobre este algoritmo, é interessante fazer uma análise dele em *traces* específicos, normalmente de algoritmos de uso científico, ou específicos para certas tarefas não realizadas por usuários comuns, como roteamento de FPGA ou até mesmo cálculos de física nuclear entre outros *traces* que possam ser analisados.

(a)

(b)

(c)

(d)

Figura 28 – Outros resultados com cache de 32KB não unificadas

A Figura 28 demonstra alguns dos resultados obtidos com alguns *traces* em memórias cache de 32 KB. A Figura 28.a apresenta os resultados do *trace* 197.parser para memórias com associatividade 2-4 way. O ganho de desempenho na taxa de faltas do algoritmo Carvalho e Martins sobre o algoritmo normal, nesse caso, foi de 11,24%, e o algoritmo Kerr e Midorikawa teve um desempenho ligeiramente inferior, com a taxa de faltas 0,16% maior que o Carvalho e Martins. O algoritmo híbrido, por sua vez, conseguiu uma melhoria de 1,33% sobre o valor do algoritmo Carvalho e Martins.

Já a simulação com o *trace* com *benchmark* de pontos flutuante, 172.mgrid,

com associatividade 4-8 way (Figura 28.b), os resultados foram um pouco diferentes. O algoritmo Carvalho e Martins conseguiu um ganho, sobre o algoritmo normal, de 18,12%. O algoritmo Kerr e Midorikawa conseguiu melhorar o desempenho em relação ao Carvalho e Martins, assim como o algoritmo híbrido conseguiu sobre o Kerr e Midorikawa, ganhos de, respectivamente 2,57% e 4,23%.

Por outro lado, a Figura 28.c (*trace* com *benchmark* de ponto flutuante 301.apsi com 8-16 way) e a Figura 28.d (*trace* de ponto flutuante 200.sixtrack com 4-8 way) apresentaram resultados muito semelhantes com a Figura 28.a. O algoritmo normal apresenta uma taxa de faltas de 14,05% e 13,74%, respectivamente, taxas estas 8,64% e 19,07% maiores que o resultado obtido com o algoritmo Carvalho e Martins. O algoritmo Kerr e Midorikawa obteve um resultado um pouco inferior ao Carvalho Martins, com perdas de 1,48% e 0,66%. Por fim, o algoritmo híbrido demonstrou mais uma vez sua eficiência sobre os outros algoritmos em memórias cache não unificadas, conseguindo obter uma melhoria na taxa de faltas de 1,62% e 3,66%, respectivamente, em relação aos resultados obtidos com as simulações realizadas com estas mesmas memórias utilizando o algoritmo de associatividade Carvalho e Martins.

(a)

(b)

(c)

(d)

Figura 29 – Outros resultados com cache de 64KB não unificadas

Para completar as análises com memórias cache utilizando outros tipos de *traces*, foram realizadas simulações com memórias cache de 64 KB no total. A Figura 29 apresenta alguns dos resultados obtidos com essas simulações. Utilizando memórias com associatividade 2-4 way, 8-16 way e, novamente, 8-16 way, a Figura 29.a (254.gap), Figura 29.c (200.sixtrack em ponto flutuante) e Figura 29.d (175.vpr place) apresentam resultados totais semelhantes, onde o algoritmo Carvalho e Martins apresenta resultados melhores que o Kerr e Midorikawa, que por sua vez apresenta desempenhos melhores que o algoritmo normal. E como esperado, o algoritmo híbrido aponta com o melhor desempenho entre essas simulações, com uma queda na taxa de faltas de 1,71%, 6,4% e 4,93%.

A Figura 29.b, ao contrário das outras três análises realizadas para 64 KB, tem o algoritmo Kerr e Midorikawa com um desempenho superior ao algoritmo Carvalho e Martins no total, com uma diferença de taxa de faltas de 5,25%, e o algoritmo híbrido, neste caso, conseguiu uma melhoria na taxa de faltas de 3,52% sobre o resultado do algoritmo Kerr e Midorikawa.

6.2 Conclusões dos resultados obtidos

Neste capítulo foi analisado o desempenho dos algoritmos de associatividade em memórias cache não-unificadas. Para fazer as análises sobre o desempenho dos algoritmos em memórias não unificadas, foram realizadas algumas simulações

por *trace*. Essas simulações foram divididas em simulações com 32 KB e 64 KB de memória total. Dentro destas divisões, foram realizadas mais três subdivisões, para memórias com associatividade 2-4 way, 4-8 way e 8-16 way (nos casos do algoritmo normal, apenas o x do x-y way primeiro valor foi utilizado) e em cada subdivisão foi simulado os resultados para os algoritmos: normal, Carvalho e Martins, Kerr e Midorikawa e o híbrido entre Carvalho e Martins para instruções e Kerr e Midorikawa para dados. Essas simulações foram realizadas para todos os *traces* da Tabela 5.

As análises se iniciaram pelos *traces* criados a partir de *benchmarks* dos programas mais utilizados normalmente, os compactadores gzip e bzip2 e o compilador gcc. Entre esses três *traces*, o gcc obteve os resultados menos animadores, com os melhores ganhos entre os programas, mas mesmo com este *trace*, o algoritmo híbrido conseguiu obter ganho sobre todos os outros algoritmos. O bzip2 e o gzip, por outro lado, mostraram ganhos significativos do algoritmo híbrido sobre todos os outros algoritmos, mostrando sua eficácia sobre os algoritmos aqui analisados.

Mas não foi somente nos *traces* de programas mais conhecidos que o algoritmo híbrido se saiu bem, em todos os *traces* analisados, tanto os de *benchmarks* de ponto flutuante quanto os de inteiros, o algoritmo híbrido se mostrou bem eficaz, com ganhos significativos, como pode ser analisado em alguns dos *traces* simulados.

Com essas comparações, pode-se observar que o comportamento do algoritmo Kerr e Midorikawa é superior ao algoritmo Carvalho e Martins para as memórias de dados, por outro lado, para as memórias de instruções, este algoritmo não consegue ser tão eficaz, tornando um algoritmo híbrido entre estes dois, a melhor opção para memórias cache não unificadas.

7 MEMÓRIAS MULTI-NÍVEIS

Neste capítulo é proposta a utilização do algoritmo de reconfigurabilidade Kerr e Midorikawa em memórias cache multi-níveis e realizada uma análise comparativa com outros algoritmos de associatividade. Para isso, será realizada uma análise experimental com modelos de memória cache, e alguns dos resultados serão apresentados para expor o desempenho dos diversos algoritmos de associatividade analisados com a utilização de memórias cache com mais de um nível. Ao fim, será apresentada uma conclusão dos resultados obtidos com as simulações realizadas com memórias cache em multi-níveis para os diversos algoritmos de associatividade.

7.1 Simulações realizadas

A utilização de outros níveis de memória é algo muito importante em um processador atual, com o conceito de diminuir cada vez mais o gargalo do acesso à memória RAM. E a sua utilização é tão importante quanto a idéia de diminuir, também nos outros níveis de memória cache, a taxa de faltas, para no final, aumentar o desempenho total do sistema.

Por esse motivo é proposta aqui a utilização de uma memória cache com associatividade reconfigurável também nos níveis subseqüentes da memória cache. Para analisar a importância de mais um nível com arquitetura reconfigurável, serão realizadas simulações comparando a arquitetura normal, Carvalho e Martin e Kerr e Midorikawa.

Para todas as simulações, foram utilizadas memórias cache L1 não-unificadas de 64 KB no total, com o algoritmo híbrido (Kerr e Midorikawa para dados e Carvalho e Martins para instruções) com associatividade 4-8 way, algoritmo de substituição LRU e tamanho de linha de 32 *bytes*. Essa configuração de memória cache L1 foi definida como padrão durante as nossas simulações para os resultados das memórias cache L2.

Definido este padrão de L1 para as simulações, foram decididas as características para as simulações das memórias L2. Atualmente, memórias cache L2 de tamanhos entre 512 KB e 1024 KB são muito comuns, e por isso, decidiu-se utilizar estes dois valores para as simulações. Para a associatividade, optou-se por

utilizar as associações 8-16 way²⁰, 16-32 way e 32-64 way (onde as duas associatividades iniciais foram escolhidas por serem muito comuns para este nível de memória cache e a terceira associatividade definida para análise).

A memória cache utilizada no L2 é do tipo unificado (arquitetura Princeton), por ser o tipo de arquitetura mais utilizada entre os desenvolvedores de processador para um nível de cache superior como este. Para estas análises de resultados, também foram realizadas simulações com todos os *traces* da Tabela 5.

7.1.1 Resultados obtidos

Os resultados obtidos com as simulações de uma memória cache L2 são muito importantes, pois uma memória cache L2 não se comporta exatamente como uma cache L1 de mesmo tamanho, pois apenas em casos de falta na L1 será realizado o acesso à memória cache L2, o que não é a mesma situação que uma memória cache L1 de mesmas características físicas.

Figura 30 – Comparação entre L1 e L2 de mesmas características

Na Figura 30, podemos observar os resultados de duas memórias cache, ambas com algoritmo normal, algoritmo de substituição LRU, tamanhos de 256 KB e associatividade 8 way com o *trace* 171.swim. A única diferença entre os resultados é

que um foi obtido com uma memória cache L1, e o outro foi obtido em uma memória cache L2, onde o L1 desta memória era uma memória cache de 64 KB, associatividade 4 way, também utilizando o algoritmo normal. Como podemos observar, os dois resultados não foram iguais, o que comprova que uma memória cache L2 possui resultados diferentes que uma memória cache L1 similar²¹, sendo interessante analisar uma possível melhoria nos resultados nos níveis subseqüentes de memória.

Para melhor visualizar os resultados obtidos com as memórias cache simuladas, dividiu-se as análises em quatro sub-categorias, uma para o compactador gzip, uma para o compilador gcc, uma para o compactador bzip2 e uma última categoria para outros *traces* científicos.

7.1.1.1 Simulações com *traces* do gzip

Inicialmente, foram realizadas simulações com o *trace* do compactador gzip, e o algoritmo Kerr e Midorikawa, assim como no capítulo anterior, apresentou resultados muito interessantes comparado com o algoritmo Carvalho e Martins e o algoritmo normal, tanto para memórias cache L2 de 512 KB, como para as memórias cache L2 de 1024 KB, como pode-se observar nos gráficos da Figura 31.

(a)

(b)

²¹ São consideradas memórias cache similares as memórias que possuem as mesmas configurações, ou seja, tamanho, algoritmos, associatividade, etc...

(c)

(d)

Figura 31 – Resultados com cache em L2 utilizando gzip

A Figura 31 apresenta alguns dos resultados dos *traces* obtidos. Na Figura 31.a, pode-se observar que com o *trace* 164.gzip random com associatividade 8-16 way, o algoritmo Kerr e Midorikawa apresentou um ganho de 3,51% sobre o algoritmo Carvalho e Martins com tamanho 512 KB, e 6,76% para memórias cache de 1 MB. Na Figura 31.b (164.gzip graphic com 16-32 way), Figura 31.c (164.gzip program com 32-64 way) e Figura 31.d (164.gzip source com 16-32 way) o ganho do algoritmo Kerr e Midorikawa sobre o segundo melhor resultado, o obtido com o algoritmo Carvalho e Martins, foi de, respectivamente para 512 KB e 1024 KB, de 4% e 3,59%, 4,44% e 5,03%, e 3,78% e 2,7%.

7.1.1.2 Simulações com *traces* do gcc

Os resultados obtidos pelo algoritmo Kerr e Midorikawa com os *traces* do compilador gcc também foram bem satisfatórios, com valores bem animadores, e desempenhos superiores que o algoritmo Carvalho e Martins e o algoritmo normal, sem nenhum tipo reconfiguração na associatividade.

Na Figura 32, pode-se observar uma análise comparativa detalhada entre os resultados do algoritmo Kerr e Midorikawa, Carvalho e Martins, e o algoritmo normal em memórias cache L2.

(a)

(b)

(c)

(d)

Figura 32 – Resultados com cache em L2 utilizando gcc

A Figura 32.a apresenta os resultados obtidos com uma memória cache L2 com associatividade 8-16 way, usando o *trace* 176.gcc 200. Com caches de 512 KB, o algoritmo normal obteve uma taxa de faltas de 5,21%, e o algoritmo Carvalho e Martins obteve um resultado 20,73% menor. O algoritmo Kerr e Midorikawa obteve um desempenho melhor ainda, com um ganho de 2,95% sobre o resultado do Carvalho e Martins. Para as simulações com memórias cache de 1024 KB, os resultados foram bastante similares, com o algoritmo Carvalho e Martins apresentando um ganho de 25,06% sobre o algoritmo normal, e o algoritmo Kerr e

Midorikawa obtendo um desempenho 3,23% melhor que o algoritmo Carvalho e Martins.

Os resultados para os outros *traces* de gcc, associatividades e tamanhos de memórias cache foram bastante similares. Na Figura 32.b (176.gcc expr com 16-32 way), o algoritmo Carvalho e Martins obteve ganhos de desempenho de 24,51% e 28,44% sobre o algoritmo normal para memórias de 512 KB e 1024 KB, e o algoritmo Kerr e Midorikawa conseguiu um desempenho ainda melhor, com ganhos de 5,61% e 3,63% sobre o algoritmo Carvalho e Martins.

Utilizando o 176.gcc 166 com associatividade 32-64 way (Figura 32.c) e o *trace* 176.gcc integ com associatividade 16-32 way (Figura 32.d), as melhorias do algoritmo Kerr e Midorikawa sobre o Carvalho e Martins foram de 6,17% e 4,65% para memórias de 512 KB e 6,39% e 3,74% para memórias de 1024 KB.

7.1.1.3 Simulações com *traces* do bzip2

O último *trace* baseado em programas de uso diário é o *trace* obtido do *benchmark* do programa de compactação bzip2 com diversos arquivos como entrada. Para todos os casos apresentados nesta seção, o algoritmo Kerr e Midorikawa conseguiu ser superior que o Carvalho e Martins, que por sua vez foi muito superior ao algoritmo normal.

(a)

(b)

(c) (d)
Figura 33 – Resultados com cache em L2 utilizando bzip2

O *trace* 256.bzip2 p7 com associatividade 8-16 way (Figura 33.a), o *trace* 256.bzip2 g7 com associatividade 16-32 way (Figura 33.b), o *trace* 256.bzip2 g9 com associatividade 32-64 way (Figura 33.c) e o 256.bzip2 s7 com 16-32 way (Figura 33.d) apresentaram ótimos resultados obtidos pelo algoritmo Kerr e Midorikawa sobre o algoritmo Carvalho e Martins, com ganhos de, respectivamente, 3,37%, 8,12%, 5,75% e 4,29% para memórias cache de 512 KB, e ganhos de 6,81%, 10,83%, 10,22% e 6,14% para memórias cache L2 de 1024 KB. Os resultados com os *traces* deste programa foram muito interessantes, o algoritmo Kerr e Midorikawa conseguiu apresentar ganhos muito expressivos sobre os outros algoritmos analisados, com casos de ganhos de até 10% sobre o valor obtido pelo algoritmo Carvalho e Martins sob as mesmas condições.

7.1.1.4 Outras simulações

Com os bons resultados obtidos pelos *traces* baseados nos *benchmarks* de programas mais conhecidos, é interessante também fazer uma análise sobre *traces* de outros *benchmarks* científicos para se observar o desempenho dos algoritmos de associatividade em memórias cache L2. Estas simulações com outros *traces* são importantes principalmente porque cada *benchmark* tem um padrão único de acesso à memória, tanto se tratando da memória de dados como a memória de instruções, o que torna a análise dos outros *traces* mais interessante.

(a)

(b)

(c)

(d)

Figura 34 – Outros resultados com cache em L2

O primeiro resultado analisado com os outros *benchmarks* científicos é o resultado obtido com a simulação do *trace* 186.crafty com associatividade 8-16 way (Figura 34.a). Para este caso, o algoritmo normal obteve uma taxa de faltas de 4,59% com memórias cache L2 de 512 KB, e o algoritmo Carvalho e Martins obteve uma taxa de 3,63%, com um ganho no desempenho de 20,86% sobre o resultado obtido com o algoritmo normal, e o algoritmo Kerr e Midorikawa obteve um resultado melhor ainda, com um ganho de 3,64% sobre o resultado do algoritmo Carvalho e Martins, com uma taxa de faltas de 3,5%. Os resultados seguiram o mesmo padrão

para memórias cache de 1024 KB, com o algoritmo Carvalho e Martins obtendo um ganho de 26,47% sobre o algoritmo normal, e o algoritmo Kerr e Midorikawa apresentou uma melhoria de 6,88% sobre o resultado obtido pelo algoritmo Carvalho e Martins.

Outro *trace* obtido de um *benchmark* que foi analisado é o 254.gap com associatividade 16-32 way (Figura 34.b). O algoritmo normal para esta simulação, obteve uma taxa de faltas de 3,11% e 2,8% com memórias de 512 KB e 1024 KB. Para esses mesmos tamanhos de memória, o algoritmo Carvalho e Martins obteve uma taxa de faltas de 2,36% e 2,11%, com uma melhoria na taxa de faltas de 24,13% e 24,48 sobre a taxa do algoritmo normal. O algoritmo Kerr e Midorikawa obteve, também para esses mesmos tamanhos, uma melhoria de 27,83% e 27,95% sobre a taxa de faltas do algoritmo normal, e uma melhoria de 4,88% e 4,59% sobre a taxa de faltas do algoritmo Cavalho e Martins.

Os outros dois *traces* selecionados para uma análise detalhada foi o 252.eon kajiya com associatividade 32-64 way (Figura 34.c) e o *benchmark* de ponto flutuante 179.art com associatividade 16-32 way (Figura 34.d). Para simulações utilizando estas configurações e com memórias de 512 KB, o algoritmo Carvalho e Martins obteve um ganho de desempenho, respectivamente, de 35,24% e 14,94% sobre o resultado obtido pelo algoritmo normal, e o algoritmo Kerr e Midorikawa obteve uma melhoria de 7,54% e 2,41% sobre o algoritmo Carvalho e Martins. Para memórias de 1024 KB, os resultados destes *traces* foram bem similares, com o algoritmo Carvalho e Martins obtendo uma melhoria de 41,3% (Figura 34.c) e 8,04% (Figura 34.d) sobre o resultado do algoritmo normal e o algoritmo Kerr e Midorikawa obteve uma taxa de faltas menor em 6,34% e 5,28% sobre o resultado do algoritmo Carvalho e Martins.

7.2 Conclusões dos resultados obtidos

Neste capítulo é analisado o comportamento dos algoritmos de associatividade (normal, Carvalho e Martins e Kerr e Midorikawa) na inclusão de memórias cache L2 à hierarquia de memória de um processador.

Utilizando memórias cache L1 de 64 KB no total (32 KB de dados com associatividade 4-8 way e algoritmo Kerr e Midorikawa e mais 32 KB de instruções com associatividade 4-8 way e utilizando o algoritmo Carvalho e Martins) e

configurações de memórias cache L2 de 512 KB e 1024 KB, com associatividades de 8-16 way, 16-32 way e 32-64 way, foi possível analisar o comportamento do algoritmo normal, Carvalho e Martins e Kerr e Midorikawa sobre a carga de diversos dos *traces* apresentados na Tabela 5.

As simulações foram bastante interessantes, apresentando resultados muito interessantes para o algoritmo Kerr e Midorikawa, conseguindo obter ganhos muito significativos no desempenho, tanto ao se comparar com algoritmo normal, quanto comparado com o algoritmo Carvalho e Martins.

Inicialmente, foi analisado o resultado dos *traces* dos compactadores gzip e bzip2, e do compilador gcc, que apresentaram resultados muito animadores, com ganhos de desempenho do algoritmo Kerr e Midorikawa sobre o algoritmo Carvalho e Martins de até 10% em alguns dos casos.

Após essas análises, foram realizadas algumas análises com outros *traces* de *benchmarks* científicos do Spec2000 não menos importantes que os analisados inicialmente. Estes *traces* também apresentaram resultados muito animadores para o algoritmo Kerr e Midorikawa, com desempenhos superiores ao desempenho obtido pelo algoritmo Carvalho e Martins, e muito superiores aos obtidos pelo algoritmo normal.

Com os resultados analisados foi possível observar o desempenho (em relação a taxa de faltas) de uma memória cache utilizando uma memória cache que foi considerada ideal, utilizando o algoritmo híbrido para a memória cache L1 não unificada, e o algoritmo para a memória cache L2, com resultados muito superiores às outras arquiteturas, independente da carga de trabalho.

8 CONCLUSÕES FINAIS E TRABALHOS FUTUROS

Neste capítulo será feita uma análise final do trabalho realizado e dos resultados obtidos durante o trabalho e ao final do capítulo, serão apresentadas as sugestões de trabalhos a serem realizados futuramente.

8.1 Conclusões finais

A constante evolução da velocidade de operação dos processadores (frequência do *clock*) está aumentando cada vez mais o gargalo referente à velocidade de acesso aos dados da memória principal, tornando cada vez mais necessária a utilização de técnicas para diminuir o efeito gerado pela alta latência do acesso aos dados e instruções na memória. Uma das técnicas mais adotadas nos processadores modernos para suprir este problema é a utilização de memórias cache.

As memórias cache conseguem melhorar muito o desempenho de um computador utilizando os princípios da localidade (espacial e temporal), mas seria interessante se elas pudessem melhorar seu desempenho, utilizando uma estratégia para analisar os acessos à memória e detectar um determinado padrão para melhor utilizar esta memória. Utilizando este intuito para melhorar o desempenho das memórias cache, diversos autores propõem a utilização de reconfiguração nas características da memória cache, entre elas, a associatividade dos conjuntos da cache.

Neste trabalho, propomos um algoritmo de reconfiguração em memórias cache, que tem como objetivo a melhoria do desempenho de memórias cache, reduzindo sua taxa de faltas. Além de propor este algoritmo, foram realizadas simulações e análises comparativas com outros algoritmos de associatividade em memórias cache.

Este trabalho baseia-se em um algoritmo de reconfigurabilidade na associatividade de memórias cache, o algoritmo Carvalho e Martins. Este algoritmo, por sua vez, baseia-se num sistema simples para obter informações das faltas e acessos aos conjuntos de uma memória cache. A partir destes valores, o algoritmo classifica os conjuntos para poder realizar a doação/recepção de linhas entre os conjuntos da cache. Durante esta análise, concluiu-se que este algoritmo possui certas deficiências no sistema de classificação. O algoritmo aqui proposto busca

melhorar estas deficiências.

Este algoritmo aqui proposto, que chamamos de algoritmo Kerr e Midorikawa, tem como idéia principal a melhoria do sistema de classificação dos conjuntos, basicamente elevando ao quadrado o número de rótulos de classificação, o que torna o sistema de classificação mais eficaz ao rotular os conjuntos. Outra adição ao algoritmo foi a idéia de doações duplas, para acelerar o processo de reconfiguração da memória cache durante uma inversão dos padrões de acessos à memória principal.

Neste trabalho, embora houvesse conhecimento da importância do *overhead* de uma possível implementação deste algoritmo em FPGA, optou-se por trabalhar na melhoria do desempenho do algoritmo com simulações em *software*, deixando este aspecto para estudos futuros. Para realizar uma implementação física em FPGA atualmente, existem algumas barreiras tecnológicas presentes nas FPGA atuais, como é o caso do alto tempo de reconfiguração que uma FPGA apresenta, o qual futuramente, com a melhoria da tecnologia utilizada, deve diminuir, tornando a utilização mais possível. Outra barreira a ser estudada é o cálculo da rotulação dos conjuntos, que necessita do desenvolvimento de um *hardware* eficiente que possa fazer esta rotulação em tempo viável.

Após a proposta de um novo algoritmo de reconfiguração em associatividade de memórias cache, foram realizadas algumas simulações para observar o comportamento do algoritmo com diversos *traces* obtidos de *benchmarks* científicos do Spec2000. Para essas execuções, foi utilizado um simulador desenvolvido durante o processo de elaboração deste trabalho, o Bobsim. Este simulador tem a capacidade de simular diversas organizações de memórias caches, gerando um arquivo de resultados com as taxas de faltas, que é a variável que precisamos diminuir em uma memória cache para que possamos melhorar o desempenho. Duas versões deste algoritmo foram desenvolvidas, sendo que a segunda versão obteve aprimorações que possibilitam o simulador ser executado por um arquivo xml de configuração, ser executado em tarefas para uma melhor utilização de processadores multicore e também a possibilidade de execução paralela em clusters, através do uso da biblioteca MPI.

Depois de definido o ambiente de simulação, foram realizadas as simulações para o estudo dos algoritmos. A primeira parte das simulações foi destinada a uma análise comparativa entre os três algoritmos estudados em memórias cache L1

unificadas. Com os resultados obtidos conseguimos observar alguns casos interessantes que influenciaram positivamente no desempenho do algoritmo Kerr e Midorikawa. O primeiro caso observado foi o padrão receptor-doador, onde o algoritmo Carvalho e Martins apresenta uma deficiência, quando comparado ao algoritmo Kerr e Midorikawa, para se readaptar a uma inversão no padrão de um conjunto de receptor para doador e vice-versa, principalmente por não poder realizar doações duplas, o que tornaria o algoritmo mais eficiente em inversões do padrão, problema que o algoritmo Kerr e Midorikawa não possui, pois pode realizar doações duplas. Outro caso estudado foi o caso da prioridade diferenciada na recepção e doação, onde o algoritmo Kerr e Midorikawa apresenta uma melhoria de desempenho sobre o Carvalho e Martins devido ao fato que ele possui um sistema melhor de classificação dos conjuntos. O terceiro caso observado foi o de uma associatividade ideal para o algoritmo Kerr e Midorikawa, a associatividade 4-8 way, que possui um equilíbrio entre conjuntos doadores e receptores, obtendo sempre os melhores resultados nas simulações.

Na segunda parte das simulações, foi observado o comportamento dos algoritmos de associatividade em memórias cache não-unificadas. Esta análise possui uma grande importância pelo fato que a maioria dos processadores atuais adotou esta arquitetura de memória cache sobre a unificada. Durante as análises realizadas, foi observado que o algoritmo Kerr e Midorikawa possui melhor desempenho para memórias cache de dados e o algoritmo Carvalho e Martins obteve melhor desempenho em memórias cache de instruções. Com essa análise, foi proposta a utilização de um algoritmo híbrido para as comparações dos resultados obtidos nas simulações. Este algoritmo híbrido apresentou ganhos significativos em todas as simulações quando comparado com o algoritmo Kerr e Midorikawa, o algoritmo Carvalho e Martins e o algoritmo normal.

A última parte das simulações foi a análise do comportamento do algoritmo normal, o algoritmo Carvalho e Martins e o algoritmo Kerr e Midorikawa em memórias cache multi-níveis. Os resultados para esta simulação foram muito animadores, com o algoritmo Kerr e Midorikawa apresentando ganhos de desempenhos significativos sobre os outros algoritmos em diversos *traces* de *benchmarks* científicos.

Após a realização de todas as simulações, foi possível chegar a uma arquitetura de memórias cache ideal baseada nos resultados observados. Seguindo

a arquitetura largamente utilizada pelos desenvolvedores de processadores (L1 e L2), concluiu-se que a configuração ideal seria uma memória cache L1 utilizando o algoritmo híbrido (capítulo 6) e utilizando associatividade 4-8 way (capítulo 5), e uma memória cache L2 utilizando o algoritmo Kerr e Midorikawa (capítulo 7). O tamanho das memórias e a associatividade da memória cache L2 não foi estudado profundamente durante as simulações, mas pensando nas limitações de *hardware* que os desenvolvedores também possuem, o caso ideal seria utilizar memórias de 64 KB (32 KB para dados e 32 KB para instruções) para L1 e 512 KB para L2, e associatividade 16-32 way para memórias cache L2.

8.2 Trabalhos Futuros

No presente trabalho, foram observados alguns resultados muito interessantes obtidos nas simulações pelo algoritmo Kerr e Midorikawa, porém, alguns itens ainda precisam ser estudados para uma melhor utilização deste algoritmo em memórias cache reais. Para trabalhos futuros, pode-se surgir:

- Um estudo aprofundado sobre a possibilidade de se obter um *hardware* de classificação mais eficiente, principalmente uma melhoria sobre o cálculo realizado durante o sistema de classificação;
- Estudar a possibilidade de desenvolver mais duas camadas de adaptação no algoritmo, uma camada de adaptatividade para realocar a posição dos ponteiros de doadores/receptores nos possíveis rótulos e outra camada de adaptatividade para melhorar a obtenção das barreiras de rotulação, utilizadas posteriormente na classificação;
- Estudar outras possibilidades de traces reais para utilizar nas simulações comparativas entre os algoritmos de associatividade, como a utilização de outra classe de aplicações, que é o caso das aplicações transacionais;
- Análise do *overhead* na implementação do algoritmo Kerr e Midorikawa em FPGA;
- Outras estratégias de reconfiguração de memórias cache como, por exemplo, a reconfiguração no tamanho de linha, algoritmo de substituição, entre outras;
- Aplicação de reconfiguração em outras partes do processador, como o conjunto de registradores, conjunto de instruções e unidades funcionais.

REFERÊNCIAS

- [1] Hennessy, J. L., Patterson, D. A. Computer Architecture: A Quantitative Approach. Morgan Kaufman, 3rd Edition, 2003.
- [2] Wulf, Wm. A., McKee, Sally A. Hitting the memory wall: Implications of the obvious. Computer Architecture News, 23(1):20–24, 1995.
- [3] Handy, Jim. The Cache Memory Book. Morgan Kaufman, 2nd Edition, 1998.
- [4] Hauck, S., DeHon, A. Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation. Morgan Kaufmann, 1st Edition, 2007.
- [5] Martins, C. A. P. S., Ordonez, E. D., Corrêa, J. B. T., Carvalho, M. B. Computação Reconfigurável: conceitos, tendências e aplicações. In: XXII Jornada de Atualização em Informática (JAI), SBC2003, Vol. 2, p.339-388, 2003.
- [6] Compton, K., Hauck, S. Reconfigurable Computing: A Survey of Systems and Software. ACM Computing Surveys, Vol. 34, No. 2, June 2002, pp. 171-210
- [7] Bansal, S., Modha, D. S. CAR: Clock with Adaptive Replacement. USENIX File and Storage Technologies (FAST), March 31-April 2, 2004, San Francisco, CA.
- [8] Jiang, S., Zhang, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June 2002, pp. 31–42.
- [9] Digilent Inc. Website: <http://www.digilentinc.com/>
- [10] Perry, D. L.. VHDL. McGraw Hill, 3rd Edition, 1998.
- [11] SimpleScalar LLC. Website: <http://www.simplescalar.com/>
- [12] Gaisler Research. Website: <http://www.gaisler.com/>
- [13] GHDL home page. Website: <http://ghdl.free.fr/>
- [14] ModelSim Website: <http://www.model.com/>
- [15] Linux Online Website: www.linux.org
- [16] Dinero IV Trace-Driven Uniprocessor Cache Simulator Website: <http://pages.cs.wisc.edu/~markhill/DineroIV/>
- [17] Brigham Young University Trace Distribution Website: <http://traces.byu.edu/>

- [18] Kerr Jr., R. B., Midorikawa, E. T. Um Algoritmo de Reconfiguração da Associatividade em Memórias Cache, In: VIII Workshop em Sistemas Computacionais de Alto Desempenho, pp. X-Y, 2007.
- [19] IBM – Mainframe – 1960's Website: <http://www-03.ibm.com/servers/eserver/zseries/timeline/1960s.html>
- [20] Liptay, J. S., Structural Aspects of the System/360 Model 85 Part II, The Cache. IBM Systems Journal, vol. 7, No. 1, 1968, pp. 15-21.
- [21] Aiken, Howard H., Hopper, Grace M. The Automatic Sequence Controlled Calculator. *Electrical Engineering*, Vol.65 No.8-9, pp.384-391, Aug 1946.
- [22] Eigenmann, R., Lilja, D. J. Von Neumann Computers. In Wiley Encyclopedia of Electrical and Electronics Engineering, Volume 23, pp. 387-400, 1999.
- [23] Haikala, I. J., Kutvonen, P. H. Split cache organizations. In Performance'84: Proceedings of the Tenth International Symposium on Computer Performance Modelling, Measurement and Evaluation, pages 459–472. North-Holland, 1985.
- [24] Intel Core 2 Extreme Processor QX9650 and Intel Core 2 Quad Processor Q9000 Series Datasheet Website: <http://www.intel.com/design/processor/datashts/318726.htm>.
- [25] Przybylski, S. A. Cache and Memory Hierarchy Design: A Performance Directed Approach. Morgan Kaufmann, 1st Edition, 1990.
- [26] Intel Pentium 4 Processor Extreme Edition on 0.13 Micron Process in the 775-Land Package Datasheet Website: <http://www.intel.com/design/Pentium4/datashts/302350.htm>.
- [27] Vaidyanathan, R., Trahan, J. L. Dynamic Reconfiguration: Architectures and Algorithms. Kluwer Academic / Plenum Publishers, 1st Edition, 2003.
- [28] Tocci, R. J., Widmer, N. S., Moss, G. L. Digital Systems: Principles and Applications. Prentice Hall, 10th Edition, 2007.
- [29] Xilinx: The Programmable Logic Company. Website: <http://www.xilinx.com/>
- [30] Altera, the Leader in Programmable Logic. Website: <http://www.altera.com/>
- [31] P. Athanas, H. Silverman. Processor Reconfiguration through Instruction-Set Metamorphosis. Computer, Volume: 26 (3), Mar. 1993, pp. 11-18.
- [32] Wazlowski, M., Agarwal, L., Lee, T., Smith, A., Lam, E., Athanas, P., Silverman, H., Ghosh, S. PRISM-II Compiler and Architecture. In: FPGAs for Custom Computing Machines (FCCM), 1993, pp. 9-16.
- [33] Hauck, S., Fry, T., Hosler, M., Kao, J. The Chimaera Reconfigurable Functional Unit. In: FPGAs for Custom Computing Machines (FCCM), 1997, pp. 87-96.

- [34] Hauser, J., Wawrzynek, J. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In: FPGAs for Custom Computing Machines (FCCM), 1997, pp. 12-21.
- [35] Wittig, R., Chow, P. OneChip: an FPGA Processor with Reconfigurable Logic. In: FPGAs for Custom Computing Machines (FCCM), 1996, pp. 126-135.
- [36] Goldstein, S., Smith, H., Moe, M., Budiu, M., Cadambi, S., Taylor, R., Laufer, R. PipeRench: a Coprocessor for Streaming Multimedia Acceleration. In: International Symposium on Computer Architecture (ISCA), 1999, pp. 28-39.
- [37] Singh, H., Lee, M., Lu, G., Kurdahi, F., Bagherzadeh, N., Filho, E. Morphosys: a Reconfigurable Architecture for Multimedia Applications. In: Brazilian Symposium on Integrated Circuits and System design (SBCCI), 1998, pp. 134-139.
- [38] Wirthlin, M., Hutchings, B. A Dynamic Instruction Set Computer. In: FPGAs for Custom Computing Machines (FCCM), 1995, pp. 99-107.
- [39] Application Acceleration with FPGA-Based Reconfigurable Computing. Website: <http://www.cray.com/products/xd1/acceleration.html>
- [40] SGI RASC Technology Website: <http://www.sgi.com/products/rasc/>
- [41] SRC Computers Implicit+Explicit Architecture Website: <http://www.srccomp.com/ImplicitExplicitArch.htm>
- [42] Jouppi, N. P. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In Proceedings of the 17th Annual International Symposium on Computer Architecture, pp. 364–373, 1990.
- [43] Kim, C., Burger, D., Keckler, S. W. An Adaptive Cache Structure for Future High-Performance Systems, IBM Austin Center for Advanced Studies Workshop, February, 2002.
- [44] Veidenbaum, A. V., Tang, W., Gupta, R., Nicolau, A., Ji, X. Adapting Cache Line Size to Application Behavior. In Int'l Conf. on Supercomputing, pages 145-154, 1999.
- [45] Alghazo, J., Akaaboune, A., Botros, N. SF-LRU Cache Replacement Algorithm. In: The 2004 International Workshop on Memory Technology, Design and Testing (MTDT'04), pages 19-24, 2004.
- [46] Dasarathan, D., Kulandaiyan, S.. Adaptive Cache Replacement Technique. In: The 9th International Conference on High Performance Computing (HiPC 2002).
- [47] Qureshi, M., Thompson, D., Patt, Y. The V-Way Cache: Demand Based Associativity via Global Replacement. In: proc. Of Int'l Symposium of Computer Architecture 2005 (ISCA-32), pp. 544-555

[48] Carvalho, M. B., Martins, C. A. P. S. Arquitetura de Cache com Associatividade Reconfigurável. In: V Workshop em Sistemas Computacionais de Alto Desempenho, pp. 50-57, 2004. Disponível em: http://www.ppgee.pucminas.br/gsd/papers/carvalho_wscad04.pdf.

[49] Dinero IV Trace-Driven Uniprocessor Cache Simulator Website: <http://pages.cs.wisc.edu/~markhill/DineroIV/>

[50] The Standard Performance Evaluation Corporation Website: <http://www.spec.org/>

[51] Kerr Jr., R. B., Midorikawa, E. T. Introdução da Computação Reconfigurável e o Uso de Ferramentas no Ensino de Arquitetura de Computadores. In: II Workshop sobre Educação em Arquitetura de Computadores, 2007. 1 CD-ROM.