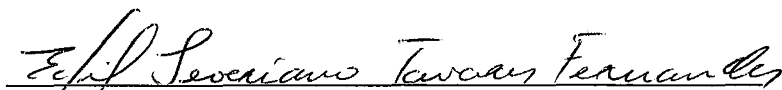
Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares

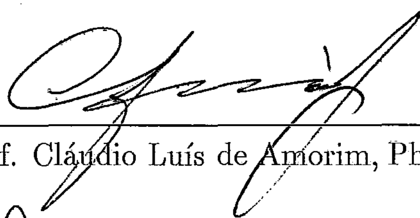
Hsing Tse Hao

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

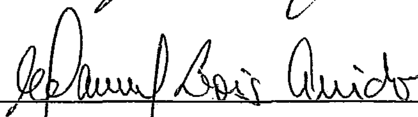
Aprovada por :



Prof. Edil Severiano Tavares Fernandes, Ph.D.
(Presidente)



Prof. Cláudio Luís de Amorim, Ph.D.



Prof. Manuel Lois Anido, Ph.D.

Rio de Janeiro, RJ – Brasil

Julho de 1993

HSING, TSE HAO

Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares [Rio de Janeiro] 1993.
XVIII, 127 p., 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1993)
Tese – Universidade Federal do Rio de Janeiro, COPPE.

1 - Arquiteturas Super Escalares
2 - Paralelismo de Baixo Nível
3 - Predição de Desvios
4 - Execução Especulativa
5 - Interrupção Precisa

I. COPPE/UFRJ
II. Título (série).

Aos meus pais e

a toda minha família

Agradecimentos

Ao professor Edil Severiano Tavares Fernandes, pela orientação competente e dedicada, pelo incentivo ao meu desenvolvimento acadêmico e pela sua amizade e confiança.

Ao Fernando Mauro Buleo Barbosa pela versão original do simulador do modelo Super Escalar, e pela sua disposição nos debates sobre a implementação do simulador e algoritmo de despacho utilizado.

Ao Alberto Ferreira de Souza, pelo simulador do i860 usado no estágio inicial da tese e pelo suporte no uso do mesmo.

Ao Eliseu Chaves Filho, pelas suas colaborações e sugestões no trabalho da tese e pela ajuda na aquisição do pacote de programas de testes *SPEC*.

Ao Delfim Xavier Martins, pelo seu companheirismo, e pelas suas incansáveis assistências nas horas mais desesperadas na implementação da tese.

Ao Ricardo Arantes, pelas facilidades em recursos que ofereceu na obtenção dos *traces* dos programas de teste.

A meus colegas Felipe Perrone, Juliana, Luciana, José Queiróz, Raquel, Nahri e Luís Carlos Quintela pela valiosa amizade.

A todos aqueles que de alguma forma contribuíram para a elaboração desta tese.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares

Hsing Tse Hao

Julho, 1993

Orientador : Edil Severiano Tavares Fernandes
Programa : Engenharia de Sistemas e Computação

Este trabalho estuda os efeitos da predição dinâmica de desvios e da interrupção precisa nos processadores Super Escalares com algoritmo de despacho de instruções dinâmico. Para suportar a análise foi implementado um modelo de processador Super Escalar com múltiplas unidades funcionais, que usa um algoritmo de despacho associativo de instruções. Utilizando simulação do tipo *Trace-driven*, avaliou-se o efeito da predição de desvio e da interrupção precisa no desempenho e na capacidade de execução especulativa do modelo. Um conjunto de programas de teste do pacote *SPEC* e utilitários de *UNIX* foram usados durante nossos experimentos.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

Effects of Branch Prediction and Precise Interrupt on the Performance of Superscalar Processors

Hsing Tse Hao

July, 1993

Thesis Supervisor : Edil Severiano Tavares Fernandes
Department : Computing and Systems Engineering

In this work, several dynamic branch prediction algorithms and precise interrupt schemes were implemented on a Superscalar model with multiple functional units, using an associative instruction dispatch algorithm. Trace-driven simulation techniques were used to assess the effects of branch prediction and precise interrupt on the performance and the speculative execution capability of the model. A set of benchmark programs from the SPEC packet and several UNIX utilities were used in our experiments.

Índice

1 Introdução **1**

1.1 Processadores Super Escalares 4

1.2 Predição de Desvios e Interrupção Precisa nos processadores Super Escalares 8

1.3 Motivação 11

1.4 Metodologia Adotada 12

1.5 Organização do Texto 13

2 Técnicas de Predição de Desvios **14**

2.1 O Custo de Desvios 15

2.2 Predição Estática de desvios 21

2.2.1 O Desvio Sempre Ocorrerá (*Always Taken Strategy*) 21

2.2.2 O Desvio Nunca Ocorrerá (*Always Not Taken Strategy*) 22

2.2.3 Previsão Baseada na Direção do Desvio 22

2.2.4 Previsão Baseada em Código de Operação 23

2.2.5 Previsão Usando *Likely/Not-Likely* Bit 23

2.3 Predição Dinâmica de Desvios 24

2.3.1 Tabela com Endereços Alvo (*Branch Target Buffer*) 26

2.3.2 Tabela com Instruções Alvo (*Branch Target Cache*) 28

2.3.3 Algoritmos de Predição Dinâmica 30

2.4 Estratégias de Desvios Usando Alterações na Seqüência de Intruções . 33

3 Esquemas de Interrupção Precisa **35**

3.1 Tipos de Estados da Máquina: Em Ordem, *Lookahead*, e Arquitetural 38

3.2 *Checkpoint Repair* 41

3.3 *History Buffer* 42

3.4 *Reorder Buffer* 44

3.5 *Future File* 48

3.6 Cancelamento da Predição Errada de Desvios Usando *Reorder Buffer* 49

3.7 Tratamento de Interrupções Internas e Externas 52

4 Modelos de Simulação **53**

4.1 A Máquina Básica 53

4.2 O Modelo Super Escalar Básico 55

4.3 Modificando o Modelo Super Escalar Básico 58

5 Método de Avaliação **62**

5.1 A Máquina Referência 62

5.2 Programas de Teste (*Benchmarks*) 63

5.3 Técnica de Simulação 64

6 Resultados de Simulação **68**

6.1 Impacto do Tamanho e da Organização do *Branch Target Buffer* . . . 69

6.2 Impacto do Algoritmo de Predição de Desvios 77

6.3 Impacto do Algoritmo de Substituição do *Branch Target Buffer* . . . 80

6.4 Impacto do Algoritmo de Alocação do *Branch Target Buffer* 83

6.5 Efeito do Tamanho do *Reorder Buffer* 86

6.6 Capacidade de Execução Especulativa 91

7 Conclusões **100**

A Dados Obtidos na Simulação **110**

A.1 Efeito do Tamanho e da Organização do BTB 110

A.2 Impacto do Algoritmo de Previsão 117

A.3 Impacto do Algoritmo de Substituição 119

A.4 Impacto do Algoritmo de Alocação 121

A.5 Efeito do Tamanho do *Reorder Buffer* 123

Lista de Figuras

2.1 Exemplo de Seqüência com Instrução de Desvio. 16

2.2 Fluxo de Instruções num *Pipeline* de Exemplo com Execução Especulativa. 17

2.3 A Penalidade Causada pelo Desvio num *Pipeline* de Exemplo. 18

2.4 Diagrama com as Penalidade de Desvio em Ciclos de Acordo com a Predição e o Resultado Efetivo de Desvio. 20

2.5 Tabela com Endereço da Instrução Alvo (*Branch Target Buffer*). . . . 25

2.6 *Branch Target Buffer*. 27

2.7 Tabela com Instrução Alvo (*Branch Target Cache*). 29

2.8 Diagrama de Estado do Esquema de Predição Dinâmica com 2 bits de Histórico. 31

3.1 Uma seqüência de instruções com despacho fora de ordem e término fora de ordem. 38

3.2 Estados em ordem, *Lookahead* e arquitetural. 39

3.3 Esquema de *Checkpoint Repair*. 41

3.4 Esquema com *History Buffer*. 42

3.5 (a) O Esquema do *Reorder Buffer*. (b) A Organização do *Reorder Buffer*. 45

3.6 Exemplo de Instruções Alocadas no *Reorder Buffer*. 47

3.7 O esquema do *Reorder Buffer* com *Future File*. 48

3.8 Restauração do Estado do Processador Após Uma Previsão Incorreta de Desvio. 50

5.1 Diagrama de Fluxo da Simulação do Tipo *Trace- Driven*. 65

6.1 Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Diff). 70

6.2 Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Li). 71

6.3 Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Whetstone). 72

6.4 Efeito do Tamanho e da Organização do BTB na Taxa de Acerto (Diff). 73

6.5 Efeito do Tamanho e da Organização do BTB na Taxa de *hit* do BTB (Diff). 74

6.6 Percentagem do Tempo de Execução em que Ocorreu o Bloqueio de Despacho de Instruções por Desvios (Li). 75

6.7 Percentagem do Tempo de Execução em que Ocorreu o Bloqueio de Despacho de Instruções por Falta de Recursos (Diff). 76

6.8 Comparação entre as Taxas de Acerto dos Algoritmos de Predição de 1 bit e de 2 bits (Whetstone). 78

6.9 Comparação entre as Taxas de Aceleração dos Algoritmos de Predição de 1 bit e de 2 bits (Grep). 79

6.10 Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Diff). 81

6.11 Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Compress). 82

6.12 Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Whetstone). 82

6.13 Comparação da Taxa de *Hit* entre os Algoritmos de Alocação *All-Branch* e *Taken* (Grep). 84

6.14 Comparação da Taxa de Acerto entre os Algoritmos de Alocação *All-Branch* e *Taken* (Grep). 85

6.15 Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Compress). 86

6.16 Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Compress). 87

6.17 Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Grep). 88

6.18 Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Grep). 89

6.19 Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Whetstone). 90

6.20 Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Whetstone). 91

6.21 Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Compress). 92

6.22 Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Espresso). 93

6.23 Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Whetstone). 95

6.24 Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Sed). 96

6.25 Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Espresso). 97

6.26 Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Whetstone). 98

6.27 Número Máximo e Médio de Instruções de Desvio Condicional alocadas no RB (Diff). 99

Lista de Tabelas

4.1 Tempos de Latência Usados no Modelo Super Escalar Básico. 57

4.2 Número das Unidades Funcionais e das *Reservation Stations* Associadas. 59

5.1 Descrição, Total de Instruções e Percentagem de Desvios dos Programas *Benchmark*. 63

A.1 Taxa de Aceleração: BTB Mapeamente Direto, *All-Branch*, 2 bits, *Miss-Taken*. 110

A.2 Taxa de Aceleração: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 110

A.3 Taxa de Aceleração: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 111

A.4 Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 111

A.5 Taxa de Acerto: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*. 111

A.6 Taxa de Acerto: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 112

A.7 Taxa de Acerto: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 112

A.8 Taxa de Acerto: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 112

A.9 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*. 113

A.10 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 113

A.11 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 113

A.12 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 114

A.13 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*. 114

A.14 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 114

A.15 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 115

A.16 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 115

A.17 Taxa de *Hit* do BTB: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*. 115

A.18 Taxa de *Hit* do BTB: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 116

A.19 Taxa de *Hit* do BTB: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 116

A.20 Taxa de *Hit* do BTB: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 116

A.21 Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*. 117

A.22 Taxa de Acerto: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*. 117

A.23 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*. 117

A.24 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*. 118

A.25 Taxa de *Hit* do BTB: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*. 118

A.26 Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 2 bits, *Random*, *Miss-Taken*. 119

A.27 Taxa de Acerto: Totalmente Associativo, *All-Branch*, 2 bits, *Random*, *Miss-Taken*. 119

A.28 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*. 119

A.29 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*. 120

A.30 Taxa de *Hit*: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*. 120

A.31 Taxa de Aceleração: 4-way Set-Associative, Taken-Branch, 2 bits, LRU, *Miss-Not Taken*. 121

A.32 Taxa de Acerto: 4-way Set-Associative, Taken-Branch, 2 bits, LRU, *Miss-Not Taken*. 121

A.33 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*. 121

A.34 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*. 122

A.35 Taxa de *Hit*: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*. 122

A.36 Taxa de Aceleração: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*. 123

A.37 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 123

A.38 Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de entradas no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 123

A.39 Número Máximo de entradas do RB ocupadas: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. . 124

A.40 Número Médio de entradas do RB ocupadas: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 124

A.41 Número Médio de Instruções Despachadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 124

A.42 Número Máximo de Instruções Executadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 125

A.43 Número Médio de Instruções Executadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. 125

A.44 Número Máximo de Desvios Condicionais no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. . 125

A.45 Número Médio de Desvios Condicionais no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*. . 126

Capítulo 1

Introdução

Durante as últimas décadas, observou-se uma busca constante no aumento de desempenho dos processadores, principalmente dos microprocessadores, que desde a sua aparição nos anos 70 vêm evoluindo num ritmo extraordinário, atingindo nos dias de hoje desempenhos antes só encontrados nos chamados supercomputadores.

Nos processadores convencionais, as instruções são executados seqüencialmente, uma após outra. Vários passos são envolvidos na execução de uma instrução, tais como a busca e a decodificação da instrução, a busca dos operandos, execução da operação especificada pela instrução, e finalmente o armazenamento do resultado da instrução. Todos esses passos são controlados pelos pulsos do relógio do processador, cujo período é chamado de tempo de ciclo do processador.

Segundo o modelo convencional acima descrito, o tempo que um processador leva para executar um programa é determinado por três fatores:

- o número de instruções requeridas para executar o programa
- o número médio de ciclos de processador necessários para executar uma instrução
- o tempo de ciclo de processador

Reduzindo um ou mais desses fatores, o desempenho de um processador pode ser melhorado. Durante toda evolução dos processadores, diversas técnicas foram criadas para alcançar esse objetivo. Por exemplo, a técnica *Pipeline*

[RAMAM77] [KOGGE81] é uma das mais utilizadas. A técnica *Pipeline* sobrepõe as etapas de execução de diferentes instruções, aproveitando o paralelismo temporal existente. O aumento de desempenho é conseguido pela redução do número médio de ciclos necessários por uma instrução, que em caso ideal pode chegar a taxa de uma instrução por ciclo.

Historicamente, os avanços da tecnologia de semicondutores e de VLSI (*Very Large Scale Integration*) sempre tiveram grandes impactos sobre a evolução dos microprocessadores. No início dos anos 70, limitadas pela tecnologia de implementação, as memórias representavam um grande gargalo para processadores, ou seja, o processador levava bem mais tempo para buscar a instrução do que para executá-la. Este fato motivou o surgimento de microprocessadores de arquitetura CISC (*Complex Instruction Set Computer*). Para aumentar o desempenho, os microprocessadores CISC procuravam alcançar dois objetivos: diminuir o número de instruções executadas e codificar densamente as instruções para minimizar o tempo de latência da memória. Os microprocessadores CISC lançam mão de instruções densamente codificadas, que requerem múltiplos ciclos para serem executadas, em detrimento do tempo de decodificação e execução dentro de processador, porém isso pode ser minimizado usando a técnica *Pipeline* para os estágios de decodificação e execução. As instruções executadas em múltiplos ciclos permitem a redução do número total de instruções necessárias e, por conseguinte, reduzem o tempo de busca das intruções.

Entretanto, durante o final dos anos 70 e o começo dos anos 80, ocorreu um grande avanço na tecnologia de fabricação de pastilhas de memória. Memórias de grande densidade e velocidade, principalmente as memórias *Caches* [SMITH82], que são memórias locais de alta velocidade implementadas perto de processadores, tornaram-se disponíveis. Isso resultou num tempo de busca de instruções da mesma ordem de grandeza do tempo de execução de instruções, portanto as suposições válidas para filosofia CISC teriam que ser revistas.

Como conseqüência, surgiram os microprocessadores RISC (*Reduced Instruction Set Computer*) [PATTR81]. Ao contrário da filosofia CISC, os processadores RISC estão preocupados em reduzir o número de ciclos necessários para

realmente executar uma instrução, uma vez que a busca de instruções já não impõe limitações tão severas como num CISC. Os processadores RISC fazem uso ostensivo da técnica *Pipeline* para reduzir o número médio de ciclos por instrução, entretanto, como suas instruções efetuam apenas operações simples, o número total de instruções pode aumentar [HENNE86]. Os estudos apresentados em [HENNE86] mostram que os processadores RISC, comparados ao processadores CISC, reduzem o número de ciclos por instrução por um fator aproximadamente de 3 a 5, enquanto aumentam o número total de instruções em 30% a 50%, produzindo portanto um ganho final no desempenho.

Embora os processadores RISC representem uma evolução em relação aos CISC, eles dependem fortemente de recursos auxiliares, tais como, um número maior de registradores, memórias *Caches* para instruções e para dados que tornam o *hardware* mais complexo e caro. Entretanto, a evolução não pára com a filosofia RISC. Na busca interminável por máquinas mais rápidas, preceitos fundamentais do modelo convencional (por exemplo, a execução seqüencial) foram questionados. Tanto no modelo CISC quanto no RISC, nos seus estágios de *Pipeline*, apenas uma instrução é executada de cada vez, uma após a outra, seqüencialmente. Ao relaxarmos essa restrição, ou seja, ao executarmos mais de uma instrução concorrentemente em cada estágio, chegaremos a um modelo novo de computação conhecido como Super Escalar. Com o objetivo de aumentar o desempenho, os processadores Super Escalares são capazes de executar várias instruções escalares simultaneamente, podendo modificar a ordem de execução das instruções. A execução concorrente permite que os processadores Super Escalares reduzam o número de ciclos por instrução aquém daquele conseguido por um processador RISC escalar com *Pipeline*, que é, no melhor dos casos, um ciclo por instrução.

Apesar do termo Super Escalar tem surgido apenas nos últimos anos, na verdade, a sua concepção teve origem nas máquinas desenvolvidas na década de 60: o CDC-6600 da Control Data Corp. [THORN64] e o 360/91 da IBM [TOMAS67] [FLYNN67] [ANDER67]. Essas máquinas já implementavam algoritmos de escalonamento capazes de despachar instruções que seriam executadas concorrentemente, todos provenientes de um mesmo programa de aplicação, para suas unidades fun-

cionais independentes. Porém a tecnologia da época não estava suficientemente madura para utilizar essas técnicas de escalonamento a um custo viável, impedindo desse modo, a aceitação da idéia entre os projetistas de processadores.

Com o estágio atual de desenvolvimento de tecnologia VLSI, onde a integração consegue atingir alguns milhões de transistores em uma única pastilha, os processadores Super Escalares tornaram-se uma realidade. Tendo em vista que a maioria dos programas que são executados nos microprocessadores são de computação escalar, e o grande potencial em desempenho demonstrado, os microprocessadores Super Escalares representam, sem dúvida alguma, o próximo passo lógico na evolução dos microprocessadores. Este fato pode ser visto pelos lançamentos recentes de microprocessadores com características Super Escalares por quase todos os grandes fabricantes do mundo, tais como: Intel, IBM, Motorola, MIPS, Sun, DEC e outros.

1.1 Processadores Super Escalares

Ao contrário das arquiteturas vetoriais e paralelas do tipo MIMD (*Multiple Instruction Multiple Data*), os processadores Super Escalares melhoram o desempenho executando simultaneamente diversas instruções escalares, que é o tipo de instrução encontrado na grande maioria de programas de aplicação. Por outro lado, as arquiteturas vetoriais e MIMD são eficientes para algumas classes de aplicações.

Nos processadores Super Escalares, a execução concorrente é possível graças à existência de unidades funcionais independentes e, mais importante, de um algoritmo de escalonamento de instruções. Provendo uma espécie de capacidade de *Lookahead* [KELLE75], o algoritmo de escalonamento de instruções examina a seqüência de instruções antecipadamente, e despacha as instruções que possam ser iniciadas concorrentemente para as unidades funcionais.

Dentro de uma seqüência de instruções existem três restrições fundamentais que limitam o desempenho de um processador Super Escalar: dependências de dado, dependências procedurais e conflito de recursos. Essas limitações são cau-

sadas pela natureza do programa de aplicação e pelas características do processador e determinam o limite superior (*Upper bound*) de desempenho que pode ser atingido pela máquina.

As dependências de dado ocorrem quando uma instrução usa o resultado produzido por uma outra (dependência verdadeira), ou quando duas instruções tentam escrever no mesmo registrador ou posição de memória (dependência de saída), ou quando uma instrução tenta escrever num registrador ou posição de memória que será lido por uma instrução precedente (anti-dependência). As dependências de saída e anti-dependências podem ser resolvidas pelos métodos de alocação de registradores na fase de compilação de programa [CHAIT81], ou pelo esquema de renomeação de registradores [KELLE75] que usam registradores adicionais alocados por *hardware*, ou finalmente pelo esquema de rotulação empregado pelo algoritmo de despacho associativo [TOMAS67].

As dependências procedurais ocorrem quando uma instrução de desvio é executada. Por causa dos desvios, o processador somente determina qual o conjunto de instruções que será executado à medida que a computação progride. Dizemos, então, que as instruções subseqüentes à instrução de desvio apresentam uma dependência procedural em relação à instrução de desvio, e que só podem ser executadas após o conhecimento do resultado de desvio.

Os conflitos de recursos ocorrem quando duas instruções tentam usar o mesmo recurso da máquina, que pode ser posição de memória, barramento, *Caches*, registradores, ou unidade funcional. Uma forma mais direta para solucionar o conflito seria atrasar uma das instruções conflitantes, porém isso prejudicaria o desempenho. No caso dos processadores Super Escalares o potencial de conflito é ainda maior do que nos processadores escalares, uma vez que múltiplas instruções são executadas a cada ciclo. Em princípio, a duplicação de recursos pode evitar o problema, mas nem sempre isso é compensador considerando o seu custo. Por exemplo, a replicação das unidades funcionais pode aumentar o número de instruções que o processador Super Escalar pode executar concorrentemente, entretanto a sua eficiência está ligada intimamente ao algoritmo de despacho de instruções e na quantidade de paralelismo existente nos programs. Portanto, um estudo cuidadoso deve

ser feito para medir a eficiência de replicação, justificando o seu custo de implementação [FERNA92a] [FERNA92b].

Num processador Super Escalar, o volume de paralelismo de um programa pode ser expresso em termos do número médio de instruções executadas por ciclo. Usualmente, esse paralelismo é determinado pelo número de dependências verdadeiras e o número de desvios executados pelo programa. As latências das operações do processador têm também uma grande influência no volume de paralelismo de um programa, pois elas podem aumentar a chance de ocorrer uma dependência verdadeira. É fundamental saber qual o volume de paralelismo dos programas a fim de justificar o uso das técnicas Super Escalares nos processadores. Trabalhos recentes em [JOUPP89], [JOHNS89], [BUTLR91] e [LAMMS92] mostram que, com a arquitetura balanceada, existe paralelismo suficiente nos programas a ser explorado pelas técnicas Super Escalares.

Como foi dito anteriormente, o algoritmo de escalonamento de instruções , que escolhe quais as instruções que devem ser despachadas e executadas em paralelo a cada ciclo e em que ordem elas são executadas, é um fator fundamental no desempenho dos processadores Super Escalares. O algoritmo de despacho de instruções fora de ordem (*Out-of-order issue*) é visivelmente melhor do que o algoritmo convencional de despacho e execução seqüencial (*In-order issue*) [JOHNS91], porque, no modelo convencional, o processador pára de decodificar as instruções toda vez que elas apresentam algum tipo de dependência de dados ou conflito de recurso em relação às instruções não terminadas. Com isso, o processador não utiliza a capacidade de *lookahead*, perdendo a chance de executar as possíveis intruções subseqüentes independentes.

Ao contrário do modelo seqüencial, o algoritmo *Out-of-order issue* separa a decodificação das instruções da execução e introduz o conceito de janela de instrução (*Instruction window*), uma espécie de *buffer* que armazena as instruções já decodificadas. Desse modo, a decodificação não pára quando uma dependência ou conflito entre instruções é encontrada, e um algoritmo de despacho seleciona instruções independentes da janela de instrução para serem executadas nas unidades funcionais. Apesar das instruções serem despachadas da janela de instrução fora da

ordem original especificada pelo programador, o algoritmo deve assegurar que a equivalência semântica do programa seja mantida.

Um algoritmo de escalonamento que permite a execução de instruções fora de ordem muito eficiente foi proposto por Tomasulo[TOMAS67], e posteriormente diversas extensões do algoritmo foram propostas em trabalhos como [BARBO92], [WEISS84], [PATTY85] e [PLESZ88]. O algoritmo de Tomasulo usa uma janela de instruções distribuída chamada de *Reservation Stations*, e junto com um esquema de rotulação (*Tagging*) e a presença de *Common Data Bus* (CDB), consegue eliminar as dependências falsas através da renomeação automática de registradores.

Um outro algoritmo que usa uma janela de instrução centralizada denominada de *Dispatch Stack* foi proposto por Torng [TORNG84]. Esse algoritmo também teve suas versões estendidas propostas em [ACOST86] e [DWYER87].

Segundo o nível de implementação, os algoritmos de escalonamento de instruções dos processadores Super Escalares podem ser classificados em dois tipos básicos: dinâmicos e estáticos. Um processador Super Escalar pode adotar escalonamento dinâmico, ou estático, ou os dois ao mesmo tempo. Os algoritmos dinâmicos são aqueles que detectam e exploram o paralelismo entre instruções em tempo de execução, e por esse motivo são implementados diretamente pelo *hardware* do processador. Devido a complexidade do *hardware* envolvido, somente os processadores mais recentes tais como i960CA da Intel [McGEA90], RS-6000 da IBM [GROHO90] [WARRE90], MC88110 da Motorola [DIEFE92] e ALPHA da DEC [ALPHA93] possuem este tipo de algoritmo.

Nos algoritmos estáticos, o escalonamento das instruções é feito antes de execução, ou seja, na fase de compilação ou otimização de código. Este tipo de algoritmo apresenta algumas desvantagens em relação aos algoritmos dinâmicos. Primeiro, eles não dispõem de informações (para detecção e exploração do paralelismo) que somente serão conhecidas em tempo de execução, e em segundo, eles dependem fortemente das técnicas de compactação [FISHE81] e otimização de códigos para atingir um bom desempenho. As máquinas VLIW (*Very Long Instruction*

Word) [FISHE84] e o processador i860 da Intel [INTEL89] são exemplos clássicos de processadores Super Escalares que utilizam este tipo de algoritmo.

Processadores com algoritmo de despacho dinâmico possuem uma grande vantagem em comparação com os processadores que empregam escalonamento estático, o que garante a sua aceitação e sucesso no mercado. Isso ocorre porque nos processadores Super Escalares com algoritmo de despacho dinâmico é possível manter a compatibilidade do código binário com os processadores anteriores. Sob o ponto de vista do programador nada precisa ser modificado para executar seu programa numa máquina Super Escalar, uma vez que o *hardware* e, em alguns casos, um módulo otimizador de código, realizam a tarefa de detecção e exploração de paralelismo. No caso das máquinas sem algoritmo de despacho dinâmico, não existe a compatibilidade em termos de *software* com nenhum processador de propósito geral existente. Além disso, devido aos diferentes tempos de latência das operações, é muito difícil garantir que duas implementações diferentes de uma mesma arquitetura com escalonamento estático sejam compatíveis a nível de código binário.

1.2 Predição de Desvios e Interrupção Precisa nos processadores Super Escalares

Com o esquema de escalonamento fora de ordem (*Out-of-order issue*), um processador Super Escalar pode executar mais de uma instrução por ciclo. Entretanto, esta taxa somente será atingida se o processador tiver um estágio de busca e decodificação de instruções capaz de fornecer uma taxa equivalente de instruções.

À primeira vista, a simples introdução de múltiplas unidades de busca e decodificação de instruções pode aumentar a taxa de busca e resolver o problema. No entanto, isso só seria válido se todas instruções de um programa fossem puramente seqüenciais. Estudos mostraram que as instruções de desvios aprarecem com uma freqüência bastante alta em programas de aplicação gerais, chegando a constituir de 15% a 30% das instruções dinâmicas de um programa [McFAR86]. As instruções de desvios quebram a seqüencialidade de endereçamento das instruções, interrompendo a busca das instruções subseqüentes até que o endereço alvo do desvio

seja conhecido. No caso dos processadores Super Escalares, a janela de instruções deve ser esvaziada rapidamente e por essa razão, o algoritmo de escalonamento não encontra mais instruções na janela para despachar para as unidades funcionais.

Os processadores RISC, *Pipelined* e Super Escalares são penalizados com uma degradação no desempenho por causa dos desvios. Em virtude de processarem (busca e execução) múltiplas instruções a cada ciclo, os processadores Super Escalares, particularmente, podem ser mais afetados por essa degradação.

Para reduzir o custo da busca e decodificação de instruções causado por desvios, e conseqüentemente amenizar a penalidade causada no desempenho do processador, esquemas de predição de desvios podem ser utilizados. Nesse esquema, existe um algoritmo que faz uma previsão do resultado do desvio (i.e., se o desvio ocorrerá ou não), durante o estágio de busca e decodificação, sem ter que esperar pelo resultado da unidade de execução que indicará se o desvio deve ser tomado ou não. Dessa forma, a busca de instrução não precisa mais ser interrompida, e a instrução sucessora pode ser rapidamente conhecida e trazida para o processador.

Algoritmos de predição de desvios podem ser classificados como estáticos ou dinâmicos. No caso estático, a previsão é feita durante a fase de compilação. Em tempo de compilação, o algoritmo de previsão utiliza algumas informações para determinar se o desvio vai ser tomado ou não. Em tempo de execução, quando uma instrução de desvio é encontrada, as instruções pertencentes ao caminho (especificado como sendo o sucessor mais provável pelo algoritmo de predição) são trazidas para a janela de instruções, impedindo desse modo, que o algoritmo de despacho seja interrompido.

Diversas estratégias podem ser utilizadas na determinação da direção do desvio, pelos algoritmos estáticos. Por exemplo, existem técnicas de predição estática que se baseiam nas características do trecho do programa, onde se encontra o desvio, para prever qual a direção mais provável (e.g., se o desvio for a última instrução de um comando repetitivo, o algorimo prediz que ocorrerá uma transferência de controle para a instrução alvejada pelo desvio).

Por outro lado, existem algoritmos estáticos que levam em considera-

ção o código de operação da instrução de desvio [LILJA88]. Por exemplo, se o código de operação da instrução de desvio especificar uma ramificação para trás (*Backward Branch*), o algoritmo de predição estática então assume que a transferência de controle ocorrerá.

Nos algoritmos dinâmicos, a previsão é levada a cabo em tempo de execução. Desse modo, o *hardware* mantém informações acerca do que ocorreu previamente com a instrução de desvio (i.e., um histórico do que ocorreu com o desvio) que está sendo executada. O algoritmo de predição de desvios (implementado na unidade de controle do processador) utiliza essas informações para prever se o desvio vai ser tomado ou não. O exemplo mais comum de predição dinâmica utiliza o *Branch Target Buffer* [LEEJK84]: uma espécie de memória *Cache* que armazena informações sobre o comportamento anterior das instruções de desvios e seus endereços alvo. Experimentos avaliando o efeito de diversas estratégias de previsão de desvios no desempenho de processadores estão apresentados em [JAMES81], [DeROS87], [FLYNN91] e [YEHTY91].

Tendo em vista que o processador Super Escalar assume que as previsões de desvios estão corretas, começando a executação das instruções pertencentes ao caminho provável de computação leva ao que é chamado de Execução Especulativa. A execução especulativa permite que a execução prossiga independentemente do término das intruções anteriores. Portanto, é de fundamental importância que a predição de desvios seja bastante precisa, pois um erro na previsão do desvio, além de causar um atraso na busca de instruções subseqüentes, pode gerar resultados incorretos na computação. Para garantir a correção no processamento, mecanismos de recuperação devem ser providenciados. Tal mecanismo de recuperação deve cancelar os efeitos causados pelas instruções executadas devido à previsão errada de desvio, e por esse motivo, torna-se necessário armazenar estados da máquina para que o processador possa reiniciar o processamento a partir da seqüência correta.

Dizemos que um processador implementa "interrupções precisas" quando o estado da máquina, visível ao sistema operacional e aos programas de aplicações, pode ser retomado ao estado que o processador teria se todas as instruções antes do ponto de interrupção já estivessem completadas. Ou seja, precisamos

conhecer exatamente o ponto de interrupção dentro do programa e o estado da máquina antes do ponto no modelo de execução seqüencial. A implementação de interrupção precisa torna-se bastante difícil nos modelos de máquina com execução fora de ordem, principalmente quando um tempo de resposta rápido for exigido, como é requerida na recuperação de previsão de desvios errada. A execução especulativa só trará ganhos no desempenho se o algoritmo de previsão de desvios fornecer uma boa taxa de acerto, e um tempo de resposta à interrupção precisa relativamente baixo.

Smith e Pleszkun [PLESZ85] publicaram um dos primeiros trabalhos com alguns métodos de implementação de interrupção precisa, tais como o *History Buffer*, o *Reorder Buffer* e o *Future File* para processadores *pipeline* com execução fora de ordem. Hwu e Patt propuseram um outro método, denominado *Checkpoint Repair* [HWUPA87]. Mais tarde, várias extensões desses métodos foram incorporadas nas máquinas com características Super Escalares em trabalhos como [GOODM87], [VAJAP87] e [TORNG92].

1.3 Motivação

As técnicas Super Escalares para implementação de processadores são, atualmente, os métodos mais utilizados na exploração do paralelismo de baixo nível. Para avaliar e melhorar o desempenho das técnicas Super Escalares, realizamos os estudos e experimentos apresentados neste trabalho.

Experimentos descritos em [FERNA93] mostraram que a previsão de desvios tem uma influência fundamental no desempenho das máquinas Super Escalares. Nesse trabalho, foi simulado um modelo de processador Super Escalar com algoritmo de despacho de instrução associativo (uma extensão do algoritmo de *Tomasulo* [TOMAS67]) e com um número grande de unidades funcionais para evitar o bloqueio no despacho de instruções por falta de recursos. Entretanto, o modelo não incorporava um esquema de predição de desvio, ou seja, ele não utilizava o conceito de execução especulativa. O despacho de instruções é bloqueado toda vez que uma instrução de desvio condicional é encontrada. Os resultados da simulação obtidos

naquele trabalho (vide [FERNA93]) mostraram que, para uma janela de instruções de tamanho 4, o algoritmo de despacho permaneceu bloqueado, aguardando pelo término das instruções de desvio de 25% a 60% do tempo total de execução dos programas de teste. Isso revela que os desvios impõem severas limitações no grau de paralelismo que pode ser atingido por um algoritmo de escalonamento dinâmico com múltiplas unidades funcionais. Posteriormente, utilizando um algoritmo de previsão de desvios onisciente, que prevê desvios com 100% de acerto, os autores observaram um aumento significativo nas taxas de aceleração (*Speedup*) na execução dos programas de teste (dependendo do programa de teste, crescimentos de 66% a 500% foram registrados. Vide [FERNA93]).

Tendo em vista as razões acima expostas, decidimos investigar o efeito no desempenho de processadores Super Escalares provocado por métodos de predição de desvios realísticos e esquemas de interrupção precisa.

1.4 Metodologia Adotada

A técnica básica utilizada no trabalho de avaliação do modelo Super Escalares foi a simulação do tipo *Trace-driven*. A simulação *Trace-driven* usa seqüências de instruções pré-determinadas, que no nosso caso são seqüências de instruções executadas por um processador escalar seqüencial. Esse processador escalar seqüencial também serve como base para obter o desempenho relativo dos nossos modelos Super Escalares.

Nos nossos experimentos, a estratégia consiste em avaliar um grande número de alternativas, comparar os desempenhos relativos e analisar o compromisso "custo × benefício" de suas implementações. Para isso, o nosso simulador do modelo Super Escalar dever permitir a exploração de um grande conjunto de especificações. Além disso, o simulador deve ser suficientemente flexível para que possamos realizar um estudo exaustivo das alternativas propostas.

Por esse motivo, incorporamos no modelo Super Escalar, desevolvido pela equipe de Arquitetura de Computadores da COPPE-Sistemas da UFRJ

[FERNA92a] [BARBO92], algoritmos de predição de desvios e de interrupções precisas. O modelo original tem sua arquitetura derivada do processador i860 da Intel Corporation [INTEL89], e reconhece o repertório de instrução do i860.

Nas avaliações de desempenho, utilizamos um conjunto de programas de teste de propósito geral. Esses programas de teste fazem parte do *SPEC Benchmarks* (*System Performance Evaluation Cooperative*) [SPEC92] , que é um pacote usado para avaliar o desempenho de *Workstations* de aplicações genéricas, e de alguns utilitários do sistema operacional *UNIX*. Os programas de teste, escritos em linguagem C, foram traduzidos para o código de máquina do i860 por um compilador comercial. Em seguida, o código foi interpretado por um simulador do processador i860, que produziu como resultado da simulação os respectivos *traces* da execução dos programas de teste. E finalmente, os *traces* foram processados pelo simulador parametrizado do modelo Super Escalar, e as medidas de desempenho foram coletadas.

Cabe ressaltar que, foi descartado neste trabalho o uso de programas de testes de tamanho pequeno que geralmente não contêm número e tipos de instrução de desvio suficientes para fornecer uma medida representativa. Todos os *traces* dos programas de teste gerados apresentam mais de um milhão de instruções, tendo seus tamanhos limitados apenas pelos recursos dos equipamentos disponíveis, durante a condução dos nossos experimentos.

Na fase final do trabalho, utilizando os resultados produzidos na nossa simulação, propomos uma configuração do modelo Super Escalar otimizado e realístico, levando-se em conta o compromisso "custo $\times$ desempenho". As principais medidas realizadas incluem: a taxa de aceleração, a taxa de acerto da previsão de desvios, o impacto na taxa de aceleração causado pela taxa de acerto e pela implementação de interrupções precisas.

1.5 Organização do Texto

Esta tese está organizada em 7 capítulos. O Capítulo 2 analisa o custo de desvios e descreve alguns métodos de predição de desvios estáticos e dinâmicos, principalmente o método de *Branch Target Buffer*. O Capítulo 3 apresenta alguns esquemas de interrupções precisas e analisa a escolha do esquema apropriado. No Capítulo 4 são descritas as diversas alternativas do modelo de simulação. A máquina de referência, os programas de teste (*Benchmarks*) e o método de simulação utilizados nos experimentos são mostrados no Capítulo 5. No Capítulo 6 encontram-se os resultados dos experimentos e sua análise. Finalmente, no Capítulo 7, apresentamos as principais conclusões do trabalho.

Capítulo 2

Técnicas de Predição de Desvios

As técnicas de predição de desvios reduzem a degradação no desempenho causada pelas instruções de desvios. Algumas informações são usadas para prever o resultado do desvio antes dele ser avaliado pelo estágio de execução do processador. Após essa predição, começa a busca antecipada da seqüência de instruções pertencentes ao caminho previsto. A predição correta elimina os ciclos perdidos na espera pelo resultado do desvio.

Quando uma instrução de desvio transfere o controle para a instrução alvo, dizemos que o desvio é tomado. No outro caso, quando o fluxo de controle passa para a instrução adjacente, dizemos que o desvio não é tomado. Quando a previsão indicar que o desvio será tomado, é necessário prever também o endereço alvo do desvio. Em alguns casos, o endereço alvo é explicitamente codificado no código de instrução, porém em outros onde há retardo na avaliação do endereço alvo, a predição torna-se necessária. O algoritmo de predição poderia, por exemplo, prever que o endereço alvo é o mesmo como ocorreu quando da última execução do desvio.

O processador pode fazer a previsão dos desvios estática ou dinamicamente. No primeiro caso, o processador usa previsões fixadas que não levam em conta a execução do programa. No caso dinâmico, o processador mantém algumas informações indicando o que ocorreu anteriormente quando da execução dos desvios, e as utiliza na predição dos novos desvios de acordo com o algoritmo de predição.

Quanto mais antecipada for a predição da direção e do endereço alvo dos desvios, menor será o atraso na busca da instrução subseqüente, e conseqüentemente melhor desempenho será conseguido. Analisamos a seguir o custo imposto pelos desvios, e os principais métodos de previsão estática e dinâmica.

2.1 O Custo de Desvios

Para executar uma instrução de desvio o processador deve determinar:

- se o desvio será tomado;
- se o desvio for tomado, qual o endereço alvo.

Os principais tipos de instruções de desvio são: desvios incondicionais, desvios condicionais, e chamada e retorno de procedimentos. Os dois primeiros são os mais freqüentemente executados. Os desvios incondicionais sempre alteram o fluxo de controle seqüencial do programa. Desvios incondicionais normalmente já contêm o endereço alvo em um campo da instrução (no caso dos processadores RISC). Em tempo de compilação ou de carga do programa esse endereço é conhecido. Portanto, os desvios incondicionais podem ser tratados pelo processador como se fossem instruções seqüenciais. A única diferença é que o contador de programa é carregado com um novo valor ao invés de ser acrescido de uma unidade.

O problema surge quando da execução de um desvio condicional: a decisão de transferir o controle está condicionada à algum resultado da computação ou ao estado do processador. O processador precisa examinar primeiro a condição para determinar o caminho correto de execução, aquele que contém a instrução do endereço alvo ou a próxima instrução seqüencial. Normalmente essa decisão depende de algum indicador de condição da máquina.

A Figura 2.1 mostra um exemplo de seqüência de instruções com um desvio condicional. A instrução *i+2* é um desvio condicional. Se a condição de desvio for falsa, as instruções subseqüentes *i+3*, *i+4*, etc. serão executadas. Caso contrário o fluxo de controle será transferido para a instrução alvo *j* e as suas sucessoras *j+1*, *j+2*, etc. Neste trabalho, vamos chamar de **instrução alvo** aquela

i
i+1
i+2 (desvio para j)
i+3
.
.
.
j
j+1
.
.

Figura 2.1: Exemplo de Seqüência com Instrução de Desvio.

que é apontada pelo campo de endereço alvo da instrução de desvio. A **instrução adjacente** é aquela que é subseqüente a uma instrução de desvio na seqüência estática de instruções, e a **instrução sucessora** é subseqüente a uma instrução de desvio na seqüência dinâmica de instruções (podendo ser uma instrução alvo ou adjacente, dependendo da condição do desvio).

Quanto à determinação do endereço alvo efetivo, as instruções de desvio podem ser grupadas em 3 tipos: endereçamento absoluto, endereçamento indexado, e endereçamento indireto. Nos processadores que utilizam endereçamento absoluto, as instruções de desvio já contêm um campo armazenando o endereço absoluto da instrução alvejada. Por esse motivo, a busca da próxima instrução não necessita de nenhuma computação ou acesso extra à memória.

No segundo tipo de endereçamento, um pouco mais complexo, o processador precisa avaliar o endereço efetivo da instrução alvo. Esse é o caso dos desvios relativos aos registradores do tipo "base", onde o deslocamento é somado com o conteúdo do registrador "base" formando o endereço efetivo da instrução alvo. Este método de endereçamento é bastante usado nos códigos de máquina relocáveis. Para agilizar o cálculo do endereço efetivo, alguns processadores possuem unidades

Com Execução Especulativa

	Estágios de Pipeline			
Ciclos	IF	ID	EX	WR
1	i			
2	i+1	i		
3	i+2	i+1	i	
4	j	i+2	i+1	i
5	j+1	j	i+2	i+1
6	·	j+1	j	i+2
7	·	·	j+1	j
8	·	·	·	j+1

Figura 2.2: Fluxo de Instruções num *Pipeline* de Exemplo com Execução Especulativa.

aritméticas dedicadas (no estágio de decodificação), outros dispõem de unidades de devios mais sofisticadas que cuidam de toda execução de desvio [DIEFE92].

O último tipo de endereçamento, chamado de endereçamento indireto, usa um ponteiro indireto para determinar o endereço alvo. O contéudo de um registrador é usado para acessar uma posição de memória que contém o endereço da instrução alvejada. Os atuais processadores RISC com arquitetura *Load-Store* geralmente não utilizam esse tipo de endereçamento para suas instruções de desvios normais (devido ao atraso do tempo de latência de memória incorrido). Por outro lado, o endereçamento indireto é bastante usado nas instruções de retorno de procedimento, pois em tempo de compilação não temos conhecimento do endereço da instrução alvo. Por essa razão, os processadores possuem um registrador *Stack Pointer* para acessar a memória indiretamente e obter desse modo o endereço alvo do desvio.

Para ilustrar o efeito causado pelos desvios condicionais, considere que o trecho de instruções da Figura 2.1 está sendo executado num processador com um *pipeline* convencional de 4 estágios: busca de instrução (IF); decodificação (ID); execução (EX); e escrita de resultados nos registradores (WR). Estamos supondo

Sem Execução Especulativa

Ciclos	IF	ID	EX	WR	
	Estágios de Pipeline				
1	i				
2	i+1	i			
3	i+2	i+1	i		
4	i+3	i+2	i+1	i	
5	i+4	i+3	i+2	i+1	
6	j	i+4	i+3	i+2	
7	j+1	j	i+4	i+3	Penalidade de Desvio
8	.	j+1	j	i+4	
9	.	.	j+1	j	
10	.	.	.	j+1	

Figura 2.3: A Penalidade Causada pelo Desvio num *Pipeline* de Exemplo.

que as instruções *i*, *i+1* e *i+2* são seqüenciais, e que *i+2* é um desvio condicional que pode transferir o controle para a instrução *j* (se o desvio for tomado) ou continuar na instrução subseqüente *i+3*.

A Figura 2.2 mostra a execução do trecho de instruções através do *pipeline* com execução especulativa. A instrução *i* entra no *pipeline* no ciclo 1 e termina no final do ciclo 4. Devido a previsão, a instrução alvo *j* inicia imediatamente depois da instrução de desvio *i+2*, sem introduzir "bolhas" (ou retardos) no *pipeline*. Se não tivéssemos execução especulativa (vide na Figura 2.3), o processador continuaria com a busca das instruções adjacentes *i+3* e *i+4*, uma vez que a decisão do desvio e o endereço alvo ainda não são conhecidos. Somente no ciclo 5, quando o desvio *i+2* sair do estágio de execução, é que o alvo *j* é buscado pelo processador e o controle é dirigido para o outro caminho. Observa-se então a penalidade de 2 ciclos introduzida pela execução incorreta das instruções *i+3* e *i+4*. Cuidados devem ser tomados para anular os efeitos causados pelas instruções incorretas no estado da máquina. No caso dos processadores Super Escalares, o exemplo acima continua válido, pois a única diferença é que, nos processadores Super Escalares, mais de uma instrução pode ser executada em cada estágio de *pipeline*.

O custo dos desvios depende da organização da máquina. A forma como a busca e a decodificação de instrução são implementadas, como é organizado o *pipeline*, e como é definida a arquitetura de instruções da máquina, são exemplos de fatores influenciando o custo das instruções de transferência de controle. Quando analisamos o custo dos desvios, podemos classificá-lo em 4 casos conforme a predição feita e o resultado real do desvio. Os 4 casos são enumerados abaixo:

1. O desvio foi previsto como não tomado, e não foi tomado realmente;
2. O desvio foi previsto como não tomado, mas na realidade foi tomado;
3. O desvio foi previsto como tomado, e a previsão estava correta;
4. O desvio foi previsto como tomado, mas na realidade não foi tomado.

Nos casos 1 e 3 a predição estava correta, e nos outros dois, a predição estava errada. Quando a predição está correta, isso não implica necessariamente que nenhuma perda no desempenho irá ocorrer. Normalmente, se o desvio é previsto como tomado, um pequeno número de ciclos de máquina é perdido na busca da instrução alvo (que está fora da sequência). Essa perda seria maior se a busca da instrução alvo ocorresse durante um *miss* na memória *cache*. Na Figura 2.4, supomos que j ciclos serão perdidos para cada desvio previsto como tomado e que será realmente tomado (caso 3).

Embora os casos 2 e 4 representem predição errada, os custos incorridos podem ser distintos. O custo do desvio previsto como tomado mas que não foi tomado (caso 4) pode ser menor do que o do desvio previsto como não tomado mas que foi tomado (caso 2): dependendo da organização do *pipeline*, as instruções adjacentes poderiam já estar disponíveis dentro do processador, devido à busca seqüencial e quando se descobre o erro na predição, essas instruções podem ser executadas imediatamente. Já no caso 2, as instruções alvo (fora da seqüência) teriam que ser buscadas após a detecção da predição errada. Os custos do caso 2 e 4 estão representados na Figura 2.4 pelos parâmetros m e k, respectivamente.

O único tipo de previsão cujo custo pode ser zero é o caso 1, quando o desvio é previsto como não tomado e a previsão estava correta. Nesse caso, conforme

		Predição	
		Não Tomado	Tomado
Resultado	Não Tomado	0	k
	Tomado	m	j

Figura 2.4: Diagrama com as Penalidade de Desvio em Ciclos de Acordo com a Predição e o Resultado Efetivo de Desvio.

especificado pela predição, as instruções adjacentes são buscadas e executadas, e o processador já estaria executando a ramificação correta sem precisar alterar nenhum estado da máquina.

Apesar da predição correta, o caso 3 pode apresentar um custo adicional. Esse custo resulta de uma predição errada no endereço alvo, provocando a busca de instruções do outro caminho. Quando a instrução de desvio termina e seu endereço alvo correto é comparado com o previsto, o processador é obrigado a cancelar os efeitos das instruções incorretas e providenciar a busca da instrução alvo correta. O custo nesse caso, seria o maior de todos, já que envolve a busca de 2 instruções, uma errada e uma correta, além do cancelamento das instruções incorretas.

Supondo que p é a probabilidade do desvio ser tomado, baseado nas suas ocorrências anteriores, então $1-p$ é a probabilidade do desvio não ser tomado. Se decidirmos que o desvio será tomado, o custo será dado por $(1-p) \times k + p \times j$. Caso contrário, o custo será dado por $p \times m$.

Cabe ressaltar que os parâmetros j, k, e m são fortemente dependentes da implementação da máquina, e que devem ser levados em consideração no

desenvolvimento de estratégias de predição de desvios para minimizar a degradação no desempenho de um processador.

2.2 Predição Estática de desvios

A predição estática de desvios também pode ser chamada de predição fixada. Utilizando uma análise do comportamento de desvios nos programas, o processador emprega uma mesma estratégia para cada tipo de instrução de desvio. Normalmente essas estratégias de previsão são implementadas diretamente no *hardware* do processador, e podem eventualmente utilizar informações fornecidas pelo compilador. A seguir, mostramos as principais estratégias de predição estática.

2.2.1 O Desvio Sempre Ocorrerá (*Always Taken Strategy*)

Uma das estratégias estáticas mais simples prevê que o desvio sempre ocorrerá, ou seja, o fluxo de controle é sempre transferido para a instrução alvo. Nesse caso, para acelerar a execução, o processador sempre faz a busca antecipada da instrução alvo. Estudos analisando o comportamento de programas indicam que os desvios (condicionais e incondicionais) são tomados em mais do que 50% dos casos. Portanto, buscar antecipadamente a instrução alvo é mais vantajoso do que buscar antecipadamente a instrução adjacente, supondo que ambos os custos sejam iguais. Nos experimentos realizados por J. E. Smith [JAMES81], foi constatada uma grande sensibilidade na taxa de acerto desse tipo de previsão, pois o programador e o compilador podem influenciar na estrutura do programa, e em conseqüência, na probabilidade dos desvios serem tomados. Dependendo do programa de teste, Smith relatou taxas de acerto variando de 57,4 a 99,6%.

O modelo 360/91 da IBM adotou esse tipo de estratégia de predição de desvio, sempre buscando antecipadamente a instrução alvo do desvio.

2.2.2 O Desvio Nunca Ocorrerá (*Always Not Taken Strategy*)

Ao contrário da estratégia anterior, o processador prevê sempre que o desvio não ocorrerá, e por esse motivo ele faz a busca antecipada da instrução adjacente. A viabilidade dessa estratégia depende intimamente do comprimento do *pipeline* do processador e da definição das funções que cada estágio do *pipeline* exercem. Segundo o estudo realizado por S. McFarling e J. Hennessy [McFAR86], apesar da percentagem dos desvios tomados ser maior do que a dos desvios não tomados, a estratégia *Not Taken* apresentou uma pequena vantagem (no desempenho global) sobre a estratégia *Taken*, num pipeline semelhante ao do processador MIPS-X^2. Isso se deve ao fato de que, na maioria das vezes, a busca da instrução alvejada (não seqüencial) tem um custo maior do que a busca da instrução adjacente. A simplicidade de implementação é uma outra vantagem em relação à estratégia anterior.

Muitos processadores reais adotaram essa estratégia de predição, entre eles o MC68020 da Motorola, o modelo VAX 11/780 da DEC e mais recentemente o i960CA da Intel, onde as instruções adjacentes ao desvio são buscadas, decodificadas e iniciadas condicionalmente.

2.2.3 Previsão Baseada na Direção do Desvio

Essa estratégia explora a freqüente ocorrência de *loops* nos programas de aplicação. Normalmente os *loops* são terminados por uma instrução de ramificação que desvia o fluxo de controle para o início do *loop*, onde uma outra instrução de desvio testa a condição para sair do *loop*. Observando essa característica, os desvios cujos endereços do alvo são menores do que o do próprio desvio (*backward branches*) são previstos como tomados. Já para os desvios cujos endereços do alvo são maiores do que o do próprio desvio (*forward branches*) são previstos como não tomados.

Se os desvios dos *loops* forem preditos corretamente, então a taxa final de acerto será alta. Entretanto, nem todos os programas possuem instruções de desvio contidas numa estrutura regular do tipo *loop*, por exemplo, nos programas que tem a maioria dos seus desvios provenientes de construções do tipo *IF-THEN-*

ELSE, a taxa de acerto será bastante afetada. Isso indica que a estratégia continua sendo muito sensível ao tipo do programa de aplicação.

Uma outra desvantagem desse método é que o endereço alvo do desvio precisa ser comparado com o conteúdo do contador de programa, para que possamos descobrir se o desvio será para frente ou não. Por esse motivo, o processo de predição é mais demorado do que o de outras estratégias.

2.2.4 Previsão Baseada em Código de Operação

Uma outra técnica de predição estática leva em consideração o código de operação da instrução de desvio. Para alguns tipos de código de operação, o processador determina se o desvio deve ser efetuado ou não. A escolha da direção é feita após a decodificação da instrução de desvio. A idéia dessa técnica consiste em explorar o comportamento repetitivo que as instruções de desvio apresentam para certos tipos de estruturas e de contruções do programa. Por exemplo, certas instruções de desvios são mais utilizadas no controle de estruturas do tipo *loop*, e outras são usadas nas construções do tipo *IF-THEN-ELSE*. Alguns modelos do System 360/370 da IBM utilizaram essa técnica de previsão.

O trabalho de J. E. Smith [JAMES81] apresenta um estudo de alguns programas escritos em FORTRAN para o computador CYBER 170. O autor observou que as instruções de desvio dos tipos "*branch if negative*", "*branch if equal*", e "*branch if greater than or equal*" normalmente provocavam uma transferência de controle. Ele aplicou a técnica para as instruções que deveriam ramificar. O resultado mostrou taxas de acerto variando de 76,2 a 99,4%.

2.2.5 Previsão Usando *Likely/Not-Likely* Bit

Essa técnica utiliza um bit, denominado *likely/not-likely* bit, no formato da instrução de desvio. Após a decodificação da instrução de desvio, o processador busca antecipadamente a instrução alvo caso o bit for igual a 1; caso contrário, a instrução adjacente é buscada. A determinação do conteúdo do bit *likely/not-likely* (ou seja, a

predição propriamente dita), fica a cargo do compilador. A eficiência dessa técnica depende da habilidade do compilador em prever os desvios. Para tal, ele leva em consideração do uso dos desvios dentro do programa ou, alternativamente, emprega algumas heurísticas. Ditzel e Mclellan [DITZE87] fizeram um estudo do processador CRISP da AT&T que utiliza essa técnica de previsão, e relataram taxas de acerto variando de 74 a 94% para uma bateria de programas.

McFarling e Hennessy propuseram um outro método, denominado *Execution Profiling* [McFAR86], para auxiliar o compilador na obteção de informações sobre os desvios que deveriam ser efetuados. O método *Profiling* consiste em seguir e anotar todos os resultados dos desvios durante a execução de um programa com um determinado conjunto de dado de entrada. O bit *likely/not-likely* de uma instrução de desvio particular indicará se o desvio, conforme o resultado do *Profiling*, deve ocorrer. Essa técnica apresenta taxa de acerto média de 85% durante a simulação de diversos programas de teste. Além disso, somente pequenas variações na taxa de acerto foram observadas quando diferentes conjuntos de dados de entrada foram usadas.

2.3 Predição Dinâmica de Desvios

Ao invés de usar critérios fixos, a predição dinâmica emprega informações do que ocorreu anteriormente com o programa, isto é, os resultados dos desvios previamente executados são utilizados na previsão. A predição dinâmica tenta adaptar-se ao programa em execução, procurando atingir a previsão ótima, e por essa razão, é muito menos sensível ao tipo do programa executado do que as técnicas estáticas. Além disso, as técnicas dinâmicas são beneficiadas com informações somente disponíveis em tempo de execução, para fazer as previsões.

A predição dinâmica de desvios explora a localidade temporal das instruções de desvio, supondo que, para uma determinada instrução de desvio, o comportamento das próximas execuções será semelhante ao que ocorreu previamente. Assim, se pudermos guardar os resultados de todos os desvios anteriormente executados, conseguiremos atingir um limite superior na taxa de acerto das previsões.

Endereço do Desvio	Endereço da Instrução Alvo	Estatística para Predição
i	x	00
j	y	01
k	z	10
⋮	⋮	⋮
m	w	11

Figura 2.5: Tabela com Endereço da Instrução Alvo (*Branch Target Buffer*).

Entretanto, devido aos recursos limitados e custo de implementação, é impossível armazenar um histórico completo de todos os desvios de um programa. Por esse motivo, torna-se necessário adotar um esquema realizável.

Um dos esquemas mais utilizados consiste em manter no processador uma pequena tabela para as instruções de desvio recentemente executadas, onde cada entrada da tabela conterá alguns bits armazenando o histórico do desvio correspondente. O processador pode acessar essa tabela associativamente (como nas memórias *cache*) ou através da técnica de *hashing*, usando o endereço da instrução de desvio como chave. Após o acesso, os bits da tabela são usados na decisão do desvio, conforme algum algoritmo de predição pré-determinado. É de se esperar que o custo de implementação depende do número de entradas dessa tabela, da sua organização e do volume de informação que ela pode guardar.

Em seguida, são apresentadas as duas implementações mais comuns desta tabela: a que contém o endereço alvo do desvio (*Branch Target Buffer*) e a que contém a instrução alvo (*Branch Target Cache*).

2.3.1 Tabela com Endereços Alvo (*Branch Target Buffer*)

O *Branch Target Buffer* (BTB) (Figura 2.5) é uma espécie de memória *cache* associada ao estágio de busca de instruções do processador. Cada entrada do BTB contém três campos: o endereço de uma instrução de desvio, o endereço alvo mais recente do desvio e informações estatísticas que permitem fazer a previsão do desvio.

O BTB funciona da sequinte forma: quando da execução de uma intrução de desvio A (no endereço i) que especifica como alvo a intrução B (no endereço x), o processador busca a instrução A e compara o seu endereço i com os conteúdos do campo "Endereço do Desvio" do BTB. Se o endereço i estiver contido numa das entradas do BTB, ou seja, se ocorrer um *hit* no BTB, a predição do desvio A será feita de acordo com a informação do campo "Estatísticas". Caso contrário, ou seja, se ocorrer um *miss* no BTB, uma das estratégias de predição estática (e.g., *Always Taken* ou *Always Not Taken*) pode ser usada para o desvio.

No caso de um *hit* no BTB, se o algoritmo de predição indicar que o desvio será tomado, então o conteúdo do campo "Endereço Alvo" será usado para acessar a instrução alvo a partir da memória principal ou da *cache*. Desta forma, o atraso na busca do alvo causado pelo cálculo do endereço efetivo será evitado. No entanto, esse atraso ocorre no caso de um *miss* no BTB. A Figura 2.6 ilustra a relação entre o estágio de busca de instruções do processador, o BTB, e a *cache* de instruções.

O processador precisa atualizar o BTB após o término da execução do desvio. Por exemplo, os bits do campo "Estatísticas" devem ser modificados conforme o resultado do desvio, e o campo "endereço alvo" tem que ser atualizado com o endereço alvo mais recente do desvio. Para atualizar o BTB, duas políticas precisam ser definidas: a política de alocação do BTB e a política de substituição do BTB. A primeira determina em que situações uma entrada do BTB deve ser alocada, enquanto que a segunda determina qual a entrada do BTB deve ser substituída por um novo desvio.

A política de alocação do BTB pode ser implementada segundo duas

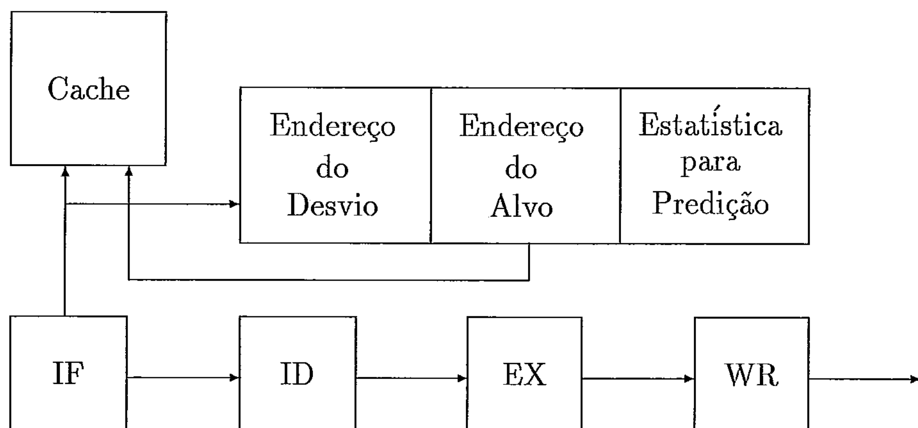


Figura 2.6: *Branch Target Buffer.*

alternativas: uma delas consiste em alocar uma entrada do BTB para cada desvio executado; na outra, a alocação é feita apenas para os desvios tomados, já que o BTB tem um tamanho finito. Na segunda alternativa, quando ocorre um *miss* no BTB, a estratégia de predição estática usada poderia ser do tipo Nunca Ocorrerá (*Always Not Taken*), pois o desvio não estando no BTB significa que ele não foi tomado na sua última execução.

Antes de alocar uma entrada no BTB, devemos selecionar a entrada que será removida. Existem basicamente duas opções nessa escolha: uma usa o algoritmo LRU global (*Least Recently Used*) que remove o desvio menos recentemente usado do BTB, e a outra usa o algoritmo randômico na seleção do desvio a ser removido do BTB. A política de substituição pode ser modificada para melhor refletir a política de alocação. Por exemplo, se os desvios não tomados não são alocados no BTB e se a política de substituição é LRU, quando um desvio não tomado já se encontra no BTB, ele pode não ser colocado no topo da lista LRU. Uma outra alternativa nessa situação seria remover o desvio não tomado do BTB para economizar espaço.

Um parâmetro importante do BTB é a taxa de *hit*. Ela expressa a

probablidade de um desvio estar armazenado no BTB durante o processamento. A taxa de *hit* é função do tamanho do BTB, da organização do BTB (associativa, mapeamento direto, ou por *hashing*), e da política de alocação e de substituição. Quanto maior for o tamanho do BTB, melhor será a taxa de *hit*. Lee e Smith [LEEJK84] mostraram que para um BTB com 256 entradas, com mapeamento direto, alocando todos os desvios (tomados e não tomados) e usando política de substituição LRU, foi obtido uma taxa de *hit* variando de 61,5 a 99,7% dependendo do programa de teste. Uma implementação real do BTB foi introduzida no computador MU-5. Holgate e Ibbett [HOLGA80] estudaram a eficiência do uso do BTB, capaz de armazenar até 8 desvios anteriormente tomados e seus respectivos endereços alvo, nessa máquina. Devido à pequena capacidade de BTB do MU-5, modestas taxas de predição correta (aproximadamente de 40 a 65%) foram conseguidas.

Dentre as alternativas de organização do BTB, as mais comuns são mapeamento direto, conjunto associativo (*set-associative*), e *hashing*, usada em muitos TLBs (*Translation Look-aside Buffers*). O acesso ao BTB *set-associative* é mais lento do que os do BTB com mapeamento direto. Apesar de possuir uma complexidade maior e um custo mais alto, a organização *set-associative* apresenta uma taxa de *hit* melhor. Os experimentos realizados por A. Smith [JAMES82] mostraram que as organizações *set-associative* e *hashing* tem praticamente o mesmo desempenho, ou seja, resultam em taxas de *hit* semelhantes. Nesse caso, a escolha deve levar em conta outros fatores como a complexidade e o custo de implementação.

2.3.2 Tabela com Instruções Alvo (*Branch Target Cache*)

Segundo a taxonomia de desvios estabelecida por H. C. Cragon [CRAGO92], quando a tabela contiver a própria instrução alvo (ao invés de apenas o endereço alvo), ela é chamada de *Branch Target Cache* (BTC). A Figura 2.7 mostra a organização do BTC, onde as instruções alvo mais recentemente executadas estão armazenadas no campo "Instruções Alvo" da BTC.

O *Branch Target Cache* atua de uma forma análoga ao *Branch Target Buffer*, com exceção de que o processador não precisa fazer a busca antecipada da

Endereço do Desvio	Instrução Alvo	Estatística para Predição
i	ld.l	00
j	addu	01
k	orh	10
⋮	⋮	⋮
m	shr	11

Figura 2.7: Tabela com Instrução Alvo (*Branch Target Cache*).

instrução alvo da memória, já que ela se encontra no BTC. Quando o processador prever que o desvio será tomado, a instrução alvo é imediatamente enviada do BTC para o estágio de decodificação sem introduzir atraso no *pipeline*, enquanto isso, o endereço alvo seria computado e a busca das instruções subseqüentes continuaria. Por essa razão, o BTC pode ser visto como uma janela alternativa de instruções. Em algumas implementações de BTC, duas ou mais instruções subseqüentes são armazenadas.

O microprocessador RISC Am29000 da Advanced Micro Devices implementa um *Branch Target Cache* com 8 entradas, cada uma contendo 4 instruções seqüenciais. No Am29000, o BTC é do tipo "*2-way* associative-set", e usa o algoritmo de substituição LRU para resolver colisões no mapeamento de uma mesma entrada. Quando uma busca de instrução não seqüencial ocorre, o BTC é consultado ao mesmo tempo que a tradução de endereço virtual é feita pela MMU (*Memory Management Unit*). Se a instrução alvo estiver no BTC, ela é passada diretamente para o *pipeline* junto com as outras três instruções adjacentes.

Um exemplo mais recente no uso do BTC pode ser encontrado no *Target Instruction Cache* (TIC) do microprocessador MC88110 [DIEFE92] da Motorola. O *Target Instruction Cache* possui 32 entradas e é totalmente associativa

(*fully associative*). Como as instruções de desvio no MC88110 têm uma latência de 2 ciclos, cada entrada do TIC contém 2 instruções da ramificação alvo que são iniciadas nesse intervalo, enquanto que as outras instruções subseqüentes são buscadas da memória em paralelo. Diefendorff e Allen [DIEFE92] mostraram que o TIC consegue atingir em média uma taxa de *hit* de 85%, executando os programas de teste do pacote SPEC.

Tanto o *Branch Target Buffer* como o *Branch Target Cache* fazem a previsão do endereço alvo ou da instrução alvo, assumindo que o resultado do desvio corrente será igual ao que ocorreu quando da execução recente do desvio. Todavia, o resultado real somente será conhecido após a execução do desvio, e existe a possibilidade do endereço previsto ser diferente do computado no caso do BTB (ou da instrução prevista ser diferente da real no caso do BTC). Embora sejam pouco freqüentes, as alterações do alvo de fato ocorrem, principalmente quando no código fonte aparecem construções do tipo *goto* computado ou *case*.

A alteração do alvo do desvio cria um custo adicional no desempenho se o algoritmo prever que o desvio será tomado, e é realmente tomado. Em outras palavras, o BTB ou BTC podem acertar na direção do desvio, mas errar no endereço (ou na instrução) alvo. Por esse motivo, toda vez que o desvio é previsto como tomado e sua execução é terminada, o alvo previsto (endereço ou instrução) deve ser comparado com o computado. Se forem diferentes, as instruções executadas indevidamente (após o desvio) devem ser canceladas, e a instrução alvo correta deve ser buscada.

Lee e Smith [LEEJK84] mediram a freqüencia de alteração do alvo de desvios para alguns programas do IBM System/370, PDP-11 e CDC6400. Para os programas do IBM System/370, eles verificaram que 1,4 a 4,4% das instruções de desvios dinâmicos tiveram seus alvos alterados. A percentagem foi de 12,0% para os programas do PDP-11, e de 2,9% para os do CDC6400.

2.3.3 Algoritmos de Predição Dinâmica

Como foi visto anteriormente, o campo "Estatística para Predição"

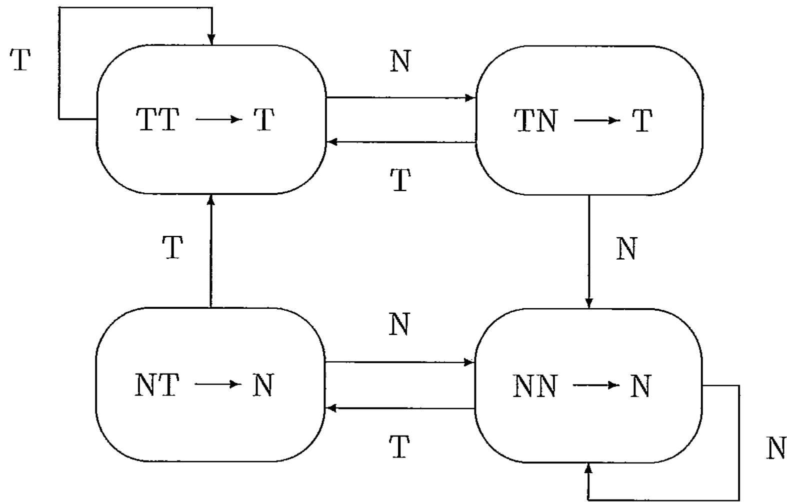


Figura 2.8: Diagrama de Estado do Esquema de Predição Dinâmica com 2 bits de Histórico.

do BTB ou do BTC guarda um histórico dos resultados de desvios executados, e utilizando esse campo, o processador faz uma previsão do que ocorrerá no futuro. Normalmente o histórico é armazenado na forma de uma seqüencia de bits, onde o bit "1" representa desvio tomado, e o valor "0" representa desvio não tomado. Dependendo da capacidade do BTB ou do BTC, o processador pode contar com n bits para prever os próximos desvios. Uma forma mais condensada para armazenar o histórico é usar os bits codificados para representar um estado do histórico, ao invés de representar diretamente a seqüência de desvios tomado ou não tomados.

Por exemplo, vamos supor que T denota um desvio tomado, e N um desvio não tomado. Dado um estado de um certo desvio em execução $S(i)$ no tempo i, a função de predição $G(S(i))$ leva ao resultado de predição T ou N. Conforme o resultado real do desvio, temos a função de atualização de estado $E(S(i), T/N) \rightarrow S(i+1)$ que transforma o estado corrente $S(i)$ para o novo estado $S(i+1)$. Portanto, o algoritmo de predição dinâmica é definida pelo número de bits (estados), a função de predição G, e a função de atualização E.

O esquema mais simples de predição dinâmica utiliza 1 bit, que guarda o resultado do desvio mais recente. Se o desvio mais recente foi tomado, o bit recebe o valor "1", caso contrário, o bit é zerado. A predição é da seguinte forma: se o bit for igual a "1", o desvio é previsto como tomado; se for igual a "0", prevê-se o contrário.

O esquema de um bit apresenta algumas ineficiências em certas situações. Por exemplo, um desvio condicional que faz o controle de um *loop* é tomado n vezes em seqüência e não tomado na $n+1$-ésima vez. A previsão vai errar na $n+1$-ésima execução do desvio, e errar uma outra vez quando o *loop* volta a ser executado. Ainda pior, para uma seqüência alternada do tipo NTNTNTNT..., o algoritmo de um bit apresentará uma taxa de 100% de erro nas previsões.

Para corrigir essa ineficiência, um esquema com 2 bits, representando as duas últimas execuções do desvio pode ser usado. Os estados, a função de predição G e a função de atualização E do esquema com 2 bits estão ilustrados no diagrama de estados da Figura 2.8. As expressões dentro dos retângulos do diagrama representam as funções de predição, que têm a forma $G(i,j) \rightarrow k$, onde i, j e $k \in \{T, N\}$. Os bits i e j representam os resultados das duas últimas execução do desvio (i.e., um estado), e o bit k é a previsão do desvio. As setas indicam a transição e atualização dos estados, e as letras ao lado das setas correspondem aos resultados reais dos desvios.

Nesse esquema de predição com 2 bits, é preciso que ocorram duas predições erradas para mudar a previsão de T para N ou de N para T. Por outro lado, dois erros também são necessários para retornar à predição anterior. No exemplo anterior do desvio no *loop*, o esquema de 2 bits só errará uma vez (na saída do *loop*). Mesmo para o caso da seqüência NTNTNTNT...., o esquema consegue acertar em 50% das previsões.

Segundo Lee e Smith [LEEJK84], apesar de ser possível variar todas as funções de predição, de atualização e o número de bits de estado para chegar a um algoritmo de predição ótima, o ganho no desempenho conseguido será muito pequeno. O algoritmo de predição de 2 bits da Figura 2.8 atingiu uma taxa de

acerto de 80,2 a 93,8% para os programas do IBM System/370, 97,8% para os do PDP-11 e 89,1% para os do CDC6400.

Outras variações na organização de BTB e no algoritmo de predição existem. Algumas implementações incluem os bits de histórico e endereço alvo do desvio na própria *cache* de instruções (ao invés de manter um BTB separado). O desempenho nesse tipo de implementação depende muito do tamanho da linha da *cache* de instrução [FLYNN91]. Yeh e Patt [YEHTY91] propuseram um algoritmo de predição dinâmica mais refinado chamado de *Two-Level Adaptive Training Branch Prediction.* O método é adaptativo porque ele altera o algoritmo de predição com base nas informações colhidas em tempo de execução. Esse algoritmo atingiu em média uma taxa de acerto de 97% para os programas do pacote SPEC. Embora o método adaptativo apresente uma taxa de acerto maior do que o esquema de predição com 2 bits, a sua implementação é complexa e o custo é elevado, uma vez que o procedimento de predição de um desvio envolve duas consultas nas tabelas contendo informações sobre desvios.

2.4 Estratégias de Desvios Usando Alterações na Seqüência de Intruções

Para minimizar os efeitos negativos no desempenho causados por desvios, técnicas como *Branch Spreading* e *Delayed Branch* são freqüentemente usadas por processadores com arquitetura RISC. Para alcançar esse objetivo, essas técnicas, normalmente implementadas no compilador, reordenam a seqüência de instruções. Além de reordenar as instruções, a técnica *Branch Spreading* [DITZE87] explora as características de uma arquitetura com instruções que modificam os indicadores de condição. Dessa maneira, as instruções de desvio, auxiliadas pelos indicadores de condição já previamente determinados, podem decidir imediatamente se o desvio deve ser efetuado. O *Delayed Branch* [GROSS82] utiliza a reordenação das instruções para mover instruções para os *delay slots* adjacentes às instruções de desvio.

Essas técnicas, outrora eficientes para os processadores RISC com despacho de apenas uma instrução por ciclo, podem não mais ser vantajosas para os

processadores Super Escalares, e representam algumas vezes, um obstáculo para o despacho de múltiplas instruções. Por exemplo, vamos supor um processador Super Escalar que possui um *pipeline* com 4 estágios e que despacha 2 instruções por ciclo. No começo de cada ciclo, podemos ter seis instruções não terminadas no *pipeline* que poderiam escrever nos registradores e alterar os indicadores de condições. Além disso, teríamos também 2 instruções a serem iniciadas. Supondo que as instruções são de três endereços (i.e., lêem dois registradores como operandos e escrevem o resultado num outro registrador), o processador precisa realizar, num único ciclo, 36 comparações entre pares de identificadores de registradores, e 12 comparações com os indicadores de condição. O número dessas comparações pode ainda aumentar se mais instruções forem despachadas a cada ciclo, ou se o comprimento do *pipeline* for aumentado. A complexidade dessas comparações inviabiliza a implementação do processador.

Situações semelhantes ocorrem quando se tenta aplicar a técnica de *Delayed Branch* a um processador Super Escalar. Para um processador que despacha 4 instruções a cada ciclo, e com uma latência de *cache* de 2 ciclos, precisamos transferir 8 instruções para os *delay slots*. A eficiência desse esquema é praticamente inalcançável, tendo em vista que, pelos experimentos realizados por McFarling e Hennessy [McFAR86], apenas em 25% dos casos o segundo *slot* foi preenchido pelas técnicas de reordenação de código. Nos outros casos, instruções NOP foram introduzidos nos *slots*.

Um dos processadores RISC Super Escalares mais atuais, o ALPHA da DEC [ALPHA93], abandonou o uso das técnicas de *Branch Spreading*, *Delayed Branch*, e outros mecanismos que possam prejudicar a implementação de um algoritmo de despacho de múltiplas intruções mais agressivo.

Capítulo 3

Esquemas de Interrupção Precisa

A eficácia da **execução especulativa** nos processadores Super Escalares se fundamenta na hipótese de que as previsões de desvios estão corretas e que não teremos interrupções. No entanto, o processador deve garantir que os resultados corretos sejam produzidos mesmo quando esta hipótese não for verdadeiro. Para garantir que os resultados estarão corretos, precisamos recuperar o estado da máquina imediatamente antes da chegada da interrupção, e reiniciar a execução da seqüência correta de instruções.

Particularmente, o modelo de execução fora de ordem dos processadores Super Escalares, onde as instruções iniciam e terminam fora da ordem especificada pelo programador, dificulta ainda mais a tarefa de restauração do estado e do reinício da execução. Normalmente, o estado da máquina consiste do contador de programa (*program counter*), dos outros registradores e da memória. Se o estado que foi resguardado, quando da ocorrência de uma interrupção, estiver consistente com o modelo seqüencial, então a interrupção é **precisa**. Mais especificamente, um estado consistente com o modelo seqüencial deve atender às seguintes condições:

- todas as instruções precedentes à instrução apontada pelo contador de programa, quando da ocorrência de uma interrupção, já foram executadas e modificaram corretamente o estado da máquina;
- todas as instruções subseqüentes à instrução apontado pelo contador de programa, quando da ocorrência de uma interrupção, ainda não foram executadas

e não modificaram o estado da máquina;

- se a interrupção foi causada por alguma condição de exceção (durante a execução de uma instrução do programa), o contexto do programa que foi resguardado deve apontar para essa instrução, e dependendo da arquitetura da máquina a instrução pode ter sido executada ou não.

Dizemos que a interrupção é **imprecisa** quando qualquer uma das três condições acima não for satisfeita.

Podemos classificar as interrupções em duas categorias:

1. Interrupções internas: ocorrem durante as fases de busca e execução de instruções. As causas desse tipo de exceção podem ser, por exemplo, tentativa de executar uma instrução inexistente, ocorrência de um *overflow*, ou uma falta de página nos processadores com memória virtual. Essas interrupções também são chamadas, de *software traps.*

2. Interrupções externas: são causadas por fontes assíncronas fora do processo em execução no processador. As interrupções de I/O e de temporizador (*timer*) são exemplos típicos dessa categoria de interrupções.

Dependendo do que se pretende no projeto da arquitetura, as interrupções podem ser implementadas como precisas ou imprecisas. Praticamente todo processador tem um conjunto de interrupções que devem ser precisas. Por exemplo, para alguns casos abaixo, a interrupção precisa é necessária ou desejável.

- no atendimento das interrupções de I/O e de temporizador, um estado preciso permite o retorno para o processo interrompido.

- na implementação de um depurador, as informações do estado preciso podem facilitar o processo de localização da instrução e das circunstâncias que causaram a condição de exceção.

- durante o tratamento de exceções aritméticas através de *software*. Utilizando as informações do estado preciso, subrotinas de tratamento podem recuperar a operação aritmética da situação de exceção.
- na reativação de um processo após o tratamento de uma interrupção por falta de página.
- para implementar, através do *software* de sistema, instruções inexistentes (i.e., "extra-code"). Dessa forma, a compatibilidade entre uma versão antiga e uma outra mais recente de um processador que possui um conjunto maior de instruções, pode ser mantida.
- para recuperar o estado do processador após uma predição de desvio errada. Essa capacidade é fundamental nos processadores Super Escalares com execução especulativa.

O problema da interrupção precisa é tão antiga quanto os primeiros computadores *pipelined*. Diferentes das máquinas seqüenciais, as máquinas *pipelined* permitem que diversas instruções estejam em diferentes fases de execução ao mesmo tempo, e que elas terminem fora da ordem seqüencial por causa das latências variadas das unidades funcionais. O IBM 360/91 é um exemplo clássico de máquina que não produzia interrupções precisas em certas situações, por exemplo, durante a execução de uma operação de ponto flutuante. Os modelos subseqüentes do IBM 360 e 370 adotaram um método de *pipeline* menos agressivo, no qual as instruções modificam os estados da máquina na estrita ordem seqüencial, garantindo desse modo que todas interrupções sejam precisas.

Máquinas de alto desempenho da década de 70 como o CDC 6600, o CDC 7600 e o CRAY-1 tinham certas condições de exceção que resultavam em interrupções imprecisas (por exemplo, nas operações de ponto flutuante). Essas máquinas optaram por um elevado grau de paralelismo e maior simplicidade de projeto, em detrimento da implementação de interrupções precisas. Por esse motivo, apenas as interrupções de I/O eram precisas. Adicionalmente, nenhuma delas implementava memória virtual, eliminando desse modo o tratamento de exceções por

Seqüência de Instruções

R3 := ... (1)
R7 := ... (2)
R8 := ... (3)
R7 := ... (4)
R4 := ... (5)
R3 := ... (6)
R8 := ... (7)
R3 := ... (8)

Figura 3.1: Uma seqüência de instruções com despacho fora de ordem e término fora de ordem.

falta de página. O CDC Cyber 180/990, lançado no começo da década 80, foi um dos primeiros computadores a utilizar um esquema de hardware, denominado *history buffer*, para garantir interrupções precisas.

3.1 Tipos de Estados da Máquina: Em Ordem, *Lookahead*, e Arquitetural

Antes de descrever os esquemas de interrupção precisa propriamente ditos, apresentamos nessa seção os conceitos de estado **em ordem**, ***lookahead*** e **arquitetural** [JOHNS91] para uma melhor compreensão sobre interrupções precisas. Para ilustrar as diferença entre esses estados da máquina, utilizaremos como exemplo um processador que inicia e termina instruções fora de ordem. A Figura 3.1 mostra uma seqüência de instruções, despachadas por esse processador, que alteram os registradores. Para simplificar o exemplo, vamos supor que os registradores são os únicos componentes do estado do processador, e que as instruções 1, 2, 3, 4 e 6 da seqüência (em negrito) já terminaram. Podemos verificar que apesar do término da instrução 6, a instrução 5 ainda se encontra em execução.

Estado em ordem	**Estado *Lookahead***	**Estado Arquitetural**
R3 := ... (1)		
R8 := ... (3)		
R7 := ... (4)		**R7 := ... (4)**
	R4 := ... (5)	R4 := ...(5)
	R3 := ... (6)	
	R8 := ... (7)	R8 := ...(7)
	R3 := ... (8)	R3 := ...(8)

Figura 3.2: Estados em ordem, *Lookahead* e arquitetural.

O estado em ordem, composto pelos mais recentes resultados produzidos pela maior seqüência contínua de instruções terminadas, corresponde ao estado obtido se o processador terminasse a execução das instruções seqüencialmente. A Figura 3.2 mostra que os resultados produzidos por três das quatro primeiras instruções da seqüência constituem o estado em ordem do processador (supondo que todas as instruções precedentes da seqüência estejam terminadas). O conteúdo de R7 atribuído pela instrução 2, não faz parte do estado em ordem, porque ele foi modificado pelo resultado mais recente produzido pela instrução 4. Embora a instrução 6 já esteja terminada, o resultado de R3 também não faz parte do estado em ordem, pois a instrução foi terminada fora de ordem (a instrução 5 ainda está em execução).

Quando as instruções terminam fora de ordem, temos então outros dois estados. O estado *lookahead* consiste dos resultados produzidos pelas instruções a partir da primeira não terminada até o final da seqüência (Figura 3.2). Por ter o nome de estado antecipado, o estado *lookahead* guarda os resultados das instruções após o ponto seguro da execução (ponto onde se sabe que todas as instruções precedentes já foram concluídas sem interrupção) , em outras palavras, nele são mantidos

resultados produzidos pelas instruções já despachadas, porém, ainda não terminadas; ou já terminadas, mas ainda sujeitas a cancelamento devido a ocorrência de uma possível interrupção. Tendo em vista que uma possível interrupção pode ocorrer, todos resultados pendentes (aqueles que ainda serão atribuídos aos registradores, e são geralmente representados por rótulos, ou "tags" em *hardware*) são mantidos no estado *lookahead.* Por exemplo, o resultado atribuído ao R3 pela instrução 6 não sobrepõe o produzido pela instrução 8, ambos os resultados devem ser acrescentados ao estado em ordem em seqüência à medida que a execução progrida. Para ilustrar como o estado *lookahead* é acrescentado ao estado em ordem, os resultados das instruções 5 e 6 passam a integrar do estado em ordem no momento em que a instrução 5 terminar sem que tenha ocorrido alguma interrupção. Ao mesmo tempo, o resultado atribuído ao R3 pela instrução 6 irá sobrepor o atribuído pela instrução 1 no estado em ordem.

O estado arquitetural consiste dos resultados das instruções já terminadas e dos resultados pendentes mais recentes de cada registrador, relativos ao final da seqüência de instruções já despachadas. Esse é o estado que deverá ser acessado pela próxima instrução da seqüência (seria a instrução 9, no exemplo da Figura 3.2) para garantir a equivalência semântica do programa. No exemplo do estado arquitetural da Figura 3.2, o resultado pendente de R3 (que será produzido pela instrução 8) sobrepõe o resultado de R3 já atribuído pela instrução 6, porque a instrução sucessora à seqüência deve ler o conteúdo mais recente de R3. Se a instrução sucessora à seqüência acessar R3 como operando e for decodificada antes do término da instrução 8, o rótulo (*tag*) associado ao novo valor do R3 será fornecido à instrução. Note que o estado arquitetural pode ser obtido combinando o estado em ordem e o estado *lookahead.*

Em seguida, serão apresentados alguns esquemas de interrupções precisas. Todos esses esquemas, de uma forma ou outra, mantêm os três estados da máquina descritos acima, e diferem apenas nos mecanismos utilizados para a obtenção e manutenção desses estados. Basicamente, os esquemas utilizam o estado em ordem para restaurar o estado da máquina quando da ocorrência de uma interrupção, mantêm o estado *lookahead* para atualização do estado em ordem à

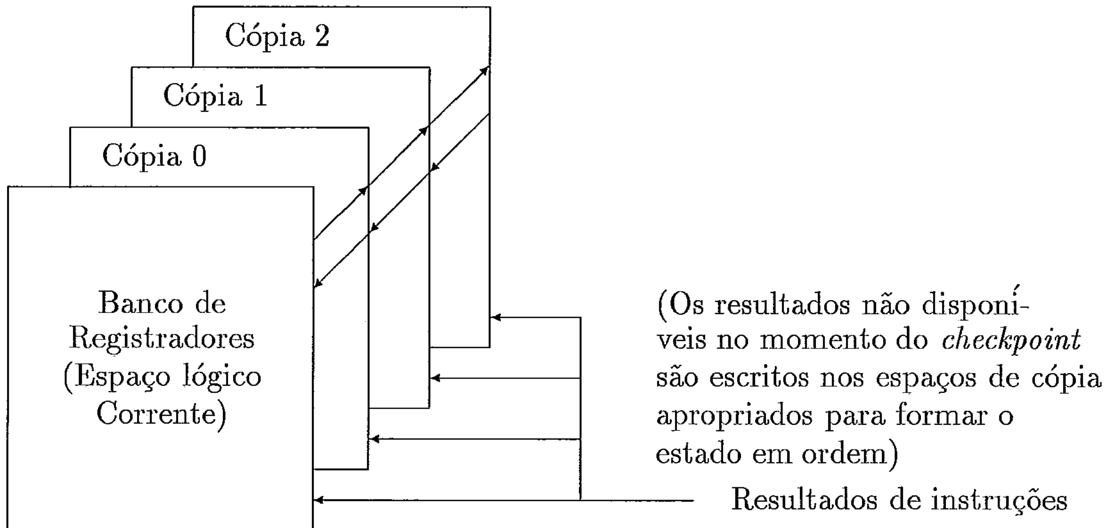


Figura 3.3: Esquema de *Checkpoint Repair.*

medida que a execução progrida, e finalmente providenciam o estado arquitetural para iniciar as novas instruções especulativamente.

3.2 *Checkpoint Repair*

Proposto por Hwu e Patt [HWUPA87], o esquema do *Checkpoint Repair* define um conjunto de espaços lógicos para a recuperação das previsões de desvio erradas e das outras exceções. Cada espaço lógico consiste do conjunto de registradores e da memória que são visíveis ao programador. De todos os espaços lógicos, apenas um é usado na execução corrente, o resto dos espaços guardam as correspondentes cópias dos estados em ordem dos pontos de execução anteriores (ver Figura 3.3).

Regularmente, durante a execução , a operação *Checkpoint* é realizada pelo processador, que copia o estado arquitetural contido no espaço lógico corrente nos outros espaços lógicos de cópia. Tendo em vista que o estado copiado não é um estado em ordem (durante o *Checkpoint*), ele precisa ser atualizado à medida que

as intruções pendentes terminem, até atingir o estado em ordem correspondente ao momento do *Checkpoint*. Os espaços lógicos atuam como uma fila FIFO: quando um novo *Checkpoint* é realizado, o espaço lógico com a cópia mais antiga é descartado.

Quando ocorre uma interrupção, as intruções precedendo o ponto de interrupção podem completar suas execuções, atualizando os estados em ordem nos espaços lógicos de cópia. A recuperação do estado da máquina é feita através do carregamento do conteúdo da cópia apropriada no espaço lógico corrente. A escolha do espaço lógico de cópia depende da distância entre o ponto de interrupção e os pontos de *Checkpoint* na seqüência do programa.

A grande quantidade de instruções de desvio nos programas obriga que a operação do *Checkpoint* seja feita também com uma freqüência bastante alta. Isso pode prejudicar sensivelmente o desempenho do processador por causa do *overhead* incorrido na operação de *Checkpoint*. A complexidade e a grande quantidade de espaço de armazenamento necessário na implementação dos espaços lógicos são outras desvantagens do esquema. É mais eficiente guardar as diferenças entre os estados em ordem, como é feito no esquema que utiliza o *History Buffer*, do que simplesmente armazenar várias instâncias completas de estados em ordem.

3.3 *History Buffer*

O *History Buffer* (Figura 3.4) foi inicialmente proposto por Smith e Pleszkun [PLESZ85] para a implementação de interrupção precisa nos processadores *pipelined* com instruções terminando fora de ordem. Nesse esquema, o estado arquitetural é mantido no banco de registradores, e o *History Buffer* mantém os itens do estado em ordem que foram sobrepostos pelos itens do estado *lookahead*. Em outras palavras, o *History Buffer* guarda os antigos conteúdos dos registradores (antes de serem sobrepostos pelos novos valores gerados pelas instruções). O estado em ordem não é mantido explicitamente nesse esquema, porém pode ser recuperado a partir dos conteúdos dos registradores e do *History Buffer*.

O *History Buffer* é organizado como uma pilha (do tipo LIFO - *Last*

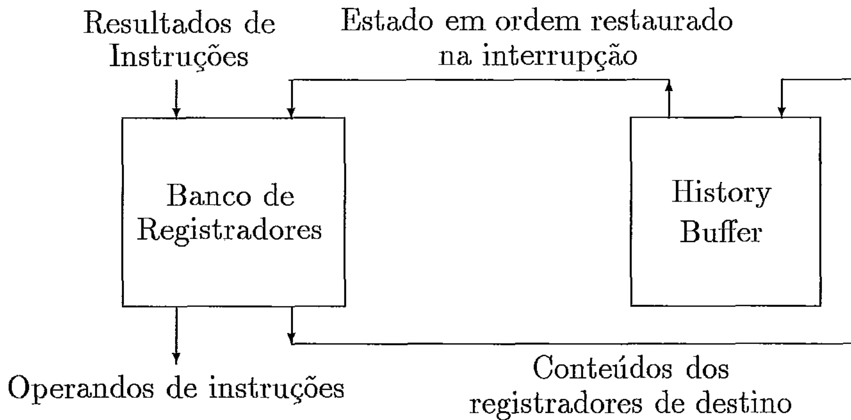


Figura 3.4: Esquema com *History Buffer.*

In First Out) de tamanho fixo. Quando uma instrução é decodificada o conteúdo do registrador destino da instrução é colocado no topo do *History Buffer*, enquanto que os outros valores que já estão no *buffer* são empurrados em direção ao fundo do *buffer*. O valor que está no fundo do *History Buffer* é descartado se a instrução associada (aquela que alocou o valor no *buffer*) terminou sem causar exceções. Se a instrução associada não estiver terminada, a decodificação de instruções é interrompida, e um novo valor é empilhado no *History Buffer* somente quando a instrução terminar.

Se uma instrução gerar algum tipo de exceção, o processador suspende o despacho de instruções e espera que todas as instruções pendentes (i.e., já iniciadas mas ainda não finalizadas) terminem e escrevam seus resultados nos registradores. Em seguida, os valores do *History Buffer* são desempilhados na ordem LIFO um por um, e escritos de volta nos registradores, até chegar o valor correspondente a instrução causadora da exceção. Se existirem diversos valores associados a um mesmo registrador, a ordem LIFO garante que o valor mais antigo será o último a ser desempilhado para o registrador, e portanto, o estado em ordem no ponto em que ocorreu a exceção será restaurado nos registradores.

O *History Buffer* é mais eficiente que o *Checkpoint Repair* em termos da implementação, porém, ainda apresenta duas desvantagens significativas em relação aos outros esquemas mostrados a seguir. Em primeiro lugar, o *History Buffer*

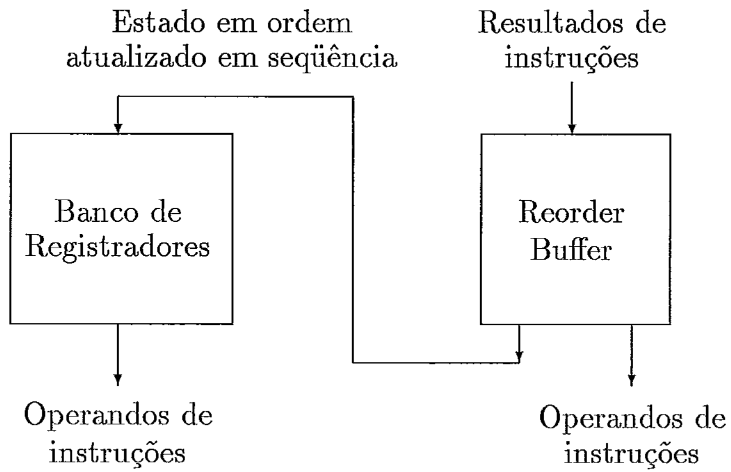
requer que o banco de registradores tenha portas de acesso adicionais para transferência do conteúdo dos registradores para o *History Buffer* na fase decodificação de instruções. Num processador Super Escalar que despacha e decodifica múltiplas intruções a cada ciclo, essas portas adicionais podem constituir um fator negativo na implementação. Em segundo lugar, quando ocorre uma previsão de desvio errada ou uma interrupção, o *History Buffer* pode levar vários ciclos de máquina desempilhando o seu conteúdo nos respectivos registradores para formar o estado em ordem, necessário pra o reinício da execução. Esses ciclos podem não ser muito críticos no reinício após uma interrupção, devido ao fato da interrupção não ser um evento freqüente, entretanto, eles são intoleráveis na reinício após uma previsão de desvio errada.

3.4 *Reorder Buffer*

No esquema de *Reorder Buffer* [PLESZ85], o banco de registradores contém o estado em ordem e o *Reorder Buffer* mantém os resultados que compõem o estado *lookahead*. O estado arquitetural não é mantido explicitamente, mas pode ser obtido combinando os estados em ordem e *lookahead*, escolhendo os valores que foram mais recentemente atualizados de cada registrador (ver Figura 3.5(a)). Basicamente, a função do *Reorder Buffer* é reordenar as instruções executadas fora de ordem, de modo que elas modifiquem o estado do processador numa ordem estritamente seqüencial.

O *Reorder Buffer* é organizada como uma fila FIFO (*First In First Out*) circular, com um ponteiro "Cabeça" que aponta para o começo da fila e um ponteiro "Livre" que aponta para a próxima entrada livre da fila. Os campos contendo diversas informações sobre as instruções despachadas são mostrados na Figura 3.5(b). Quando uma instrução é decodificada, a entrada livre é alocada à instrução, e em seguida, o ponteiro "Livre" é avançado em uma posição. No fim da execução da instrução, o seu resultado é enviado ao *Reorder Buffer* para a posterior escrita no registrador.

Na alocação de uma entrada do *Reorder Buffer*, o campo "Registrador



(a)

	No. de Entrada	Reg. Destino	Resultado	Exceção	Válido	PC
	2					
Cabeça →	3	R5	T2	n.d.	0	14
	4	R8	T6	n.d.	0	15
Livre →	5					
	⋮	⋮	⋮	⋮	⋮	⋮

n.d. - Ainda não disponível

(b)

Figura 3.5: (a) O Esquema do *Reorder Buffer*. (b) A Organização do *Reorder Buffer*.

Destino" é carregado com a identificação do registrador destino da instrução decodificada. O campo "Resultado" recebe o rótulo da unidade funcional que produzirá o resultado da instrução, se um algoritmo de despacho associativo do tipo Tomasulo [TOMAS67] for usado. Com o rótulo, o resultado da execução pode ser escrito na entrada correta do *Reorder Buffer* através de uma busca associativa. Se o bit do campo "Válido" for igual a zero, então a instrução associada não terminou. Esse bit recebe o valor "1" após o término da instrução e da escrita do seu resultado no *Reorder Buffer.* No campo "PC" é armazenado o valor do contador de programa da instrução decodificada. Se a instrução gerar uma interrupção, o conteúdo do campo "PC" pode ser carregado no registrador PC do processador, reiniciando a execução a partir do ponto de interrupção.

Além do resultado da execução, a condição de exceção também é enviada para o *Reorder Buffer* ao término da instrução. Se a instrução causou uma interrupção, o campo "exceção' recebe o valor "1". Quando a entrada apontada pelo ponteiro "Cabeça" contiver um resultado válido (i.e., a instrução associada foi concluída), o processador verifica se o bit "exceção" é igual a "1" ou não. Se o bit for igual a "0" o resultado é escrito no registrador indicado pelo campo "Registrador Destino" e o ponteiro "Cabeça" é avançado em uma posição. Caso contrário, todos os conteúdos do *Reorder Buffer* são descartados, e o processador reinicia a execução com o estado em ordem contido nos registradores. Como o tamanho do *Reorder Buffer* é finito, o despacho das instruções é interrompido quando não há mais entradas livres no *buffer.* A situação persiste até que alguma instrução termine e libere uma entrada para uma nova alocação.

Para melhor ilustrar o mecanismo de funcionamento do *Reorder Buffer*, considere o trecho da seqüência de instruções da Figura 3.6. A instrução que está no endereço 14 é uma instrução de multiplicação, cujo registrador destino é o R5 e com um tempo de latência de 6 ciclos. A instrução sucessora é uma adição com um tempo de latência de um ciclo, e por esse motivo terminará antes da instrução de multiplicação. A alocação dessas instruções está mostrada na Figura 3.5(b). Note que a instrução de multiplicação é alocada antes da instrução de soma, embora termine posteriomente. O resultado da adição somente será transferido para o regis-

PC	Instrução	Tempo de Latência
.		
.		
13		
14	R5 := R1 * R2	6 ciclos
15	R8 := R3 + R4	1 ciclo
.		
.		
.		

Figura 3.6: Exemplo de Instruções Alocadas no *Reorder Buffer*.

trador R8 depois da transferência do resultado da multiplicação para o registrador R5.

Para manter o estado preciso em relação à memória, uma solução seria alocar também as instruções de *Store* (escrita de memória) no *Reorder Buffer*, da mesma forma como as outras instruções. Desse modo, a instrução de *Store* é despachada e fica retida numa fila dentro da unidade funcional de *Load/Store*, esperando o momento adequado para ser realmente efetivada. Quando o ponteiro "Cabeça" do *Reorder Buffer* chegar à entrada contendo a instrução *Store*, isso significa que todas as instruções precedentes terminaram sem causar nenhuma exceção, e portanto, o *Store* pode ser finalmente executado.

A grande desvantagem do *Reorder Buffer* é a necessidade da busca associativa para achar os resultados mais recentes dos registradores dentro do *Reorder Buffer*, para a formação do estado arquitetural. Se para um certo registrador, existirem vários resultados dentro do *Reorder Buffer*, achar o mais recente entre eles significa que a busca associativa ainda precisa considerar a ordem das instruções. Isso deixa a implementação do *Reorder Buffer* bastante complexo. Excetuando esse problem, o *Reorder Buffer* elimina todas as desvantagens do *History Buffer*, tais como a necessidade de portas adicionais de leitura para os registradores, e a demora

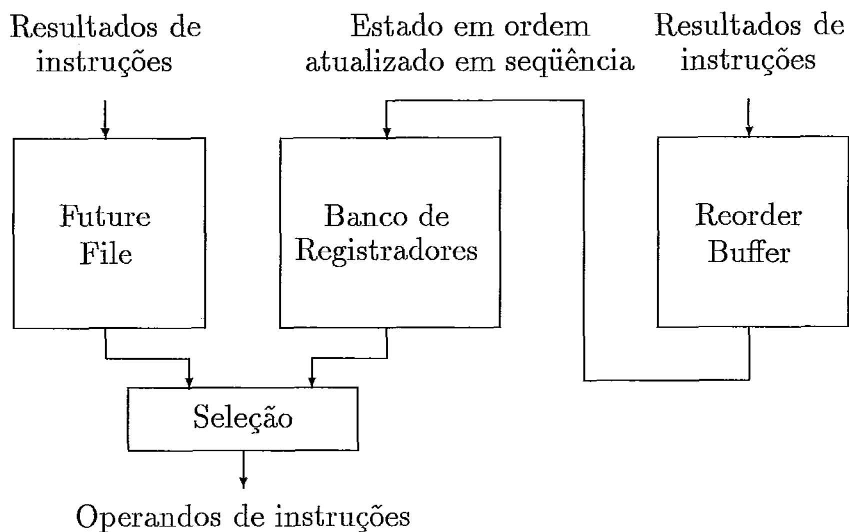


Figura 3.7: O esquema do *Reorder Buffer* com *Future File.*

na recuperação do estado em ordem. O *Reorder Buffer* mantém sempre o estado em ordem no banco de registradores, e pode descartar o estado *lookahead* contido num único ciclo, permitindo desse modo o imediato reinício.

3.5 *Future File*

O problema da busca associativa do *Reorder Buffer* pode ser evitado pelo *Future File* [JOHNS91], Essa estrutura atua junto com o *Reorder Buffer* e tem uma organização idêntica à do banco de registradores (por exemplo, se o banco tiver 32 registradores designados de R0 a R31, o *Future File* terá os mesmos registradores R0 a R31). O nome "Future" é atribuído a estrutura porque ela mantém o estado arquitetural, que é uma versão adiantada do estado em relação ao estado em ordem. O *Reorder Buffer* e o banco de registradores continuam contendo o estado *lookahead* e o estado em ordem, respectivamente, e funcionam da mesma forma como no esquema do *Reorder Buffer* descrito na seção anterior. A Figura 3.7 mostra o diagrama do *Future File.*

No esquema de *Future File*, a busca dos operandos das instruções

não é mais feita a partir do *Reorder Buffer*, que tem agora a única função de reordenar as instruções. Durante a decodificação de instruções, o processador procura os operandos no *Future File* e no banco de registradores, aquele que possuir o valor mais recente do registrador será acessado. Vale ressaltar que no *Future File* nem sempre temos o valor do registrador, mas sim rótulos ("*tags*") que representam resultados das instrução pendentes. Por outro lado, mantendo sempre o estado em ordem, o banco de registradores possui somente valores válidos, sem nenhum rótulo. Quando um valor ou um rótulo é escrito na entrada do *Future File* associada à um certo registador, a entrada é marcada como sendo a mais recente. Isso pode ser feita simplesmente posicionando um bit "Mais Recente" associado a cada entrada. As instruções subseqüentes que acessam esse registrador passarão a usar o conteúdo dessa entrada. Quando uma instrução terminar, o seu resultado é transferido para a entrada do *Future File* correspondente ao seu registrador destino, exceto se esse resultado já tiver sido sobreposto pelo resultado de alguma instrução subseqüente. Nesse caso, o resultado é simplesmente descartado, pois ele não mais faz parte do estado arquitetural.

Quando ocorre uma exceção, o processador atualiza, a partir do *Reorder Buffer* o estado em ordem nos registradores até o ponto de exceção, e descarta todos os conteúdos do *Future File* e o restante dos conteúdos do *Reorder Buffer*. Em seguida, a execução é reiniciada com o estado em ordem mantido nos registradores.

O esquema de *Future File* permite também uma rápida recuperação e reinício da execução, sem a busca associativa do *Reorder Buffer*. Entretanto, tudo isso é conseguido através de um espaço de armazenamento adicional (o próprio *Future File*).

3.6 Cancelamento da Predição Errada de Desvios Usando *Reorder Buffer*

Para reduzir ao mínimo a penalidade imposta pelas previsões de desvio erradas, os processadores Super Escalares devem possuir meios rápidos de restaurar o estado consistente e reiniciar a execução. Este requisito descarta a pos-

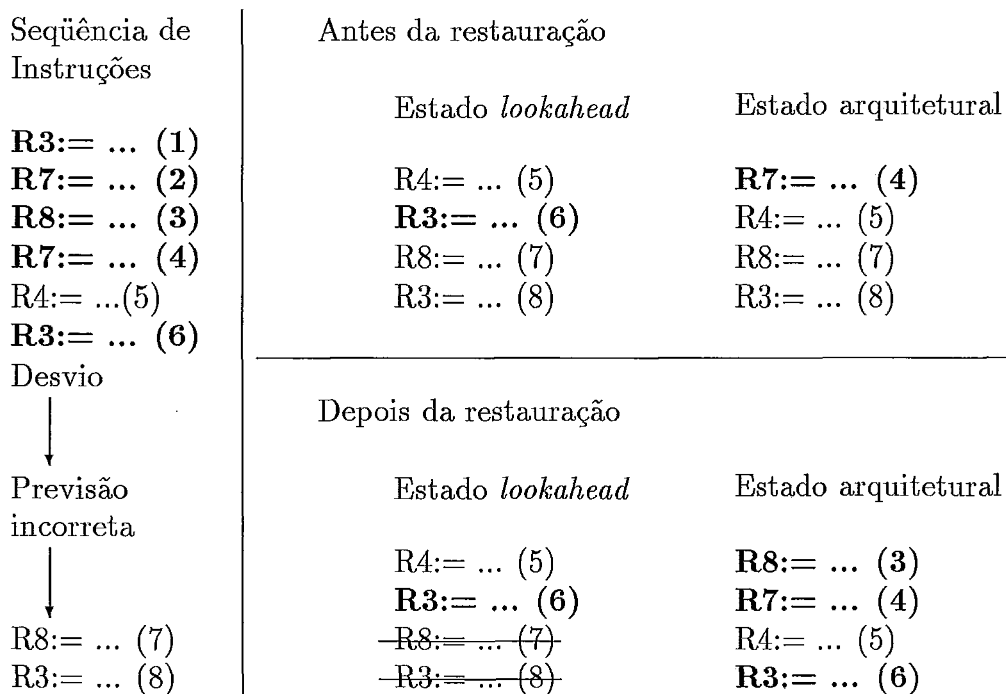


Figura 3.8: Restauração do Estado do Processador Após Uma Previsão Incorreta de Desvio.

sibilidade do uso de técnicas de *software* na solução do problema. Os mecanismos de recuperação e reinício têm que ser implementados diretamente pelo *hardware* do processador.

Com as instruções terminando fora de ordem, a tarefa do processador de distingüir as instruções que precedem o desvio das que o sucedem, pode ser bastante complexa. Se o desvio for incorretamente predito, os resultados gerados pelas instruções precedentes devem ser mantidos, e os gerados pelas instruções sucessoras devem ser cancelados. Para mostrar como o processador deve restaurar o estado após uma previsão errada de desvio, tomamos o exemplo da seqüência de instruções da Figura 3.1, e introduzimos uma instrução de desvio predita incorretamente entre as instruções 6 e 7 da seqüência. A Figura 3.8 mostra essa seqüência, e os estados *lookahead* e arquitetural antes e depois da restauração. As instruções 7 e 8 foram executadas indevidamente, portanto seus resultados produzidos em R8 e R3, respectivamente, devem ser cancelados. Não é correto cancelar todos os itens do estado *lookahead*, pois ele pode conter instruções não terminadas que precedem o desvio, e cujos resultados devem ser preservados.

Para localizar a posição exata do ponto de desvio, e por conseguinte, distingüir as instruções precedentes das sucessoras dentro do *Reorder Buffer*, a melhor solução seria alocar uma entrada no *Reorder Buffer* para cada instrução de desvio. No caso do esquema de *Reorder Buffer* sem *Future File*, quando o processador detecta uma previsão incorreta, ele descarta todas as entradas subseqüentes à entrada alocada ao desvio, e reinicia imediatamente a execução, uma vez que o estado arquitetural está disponível no *Reorder Buffer* e pode ser obtido através da busca associativa. Esse esquema exige que o *Reorder Buffer* tenha a capacidade de descartar seletivamente algumas entradas.

O caso do *Reorder Buffer* com *Future File* é um pouco mais complicado. O *Future File* não faz distinção entre resultados corretos e incorretos, logo, ele contém apenas o estado arquitetural do ponto corrente da execução, e não o estado arquitetural imediatamente antes do ponto do desvio. Para o correto reinício da execução, o processador precisa esperar que todas as instruções correspondentes às entradas que precedem à do desvio (dentro do *Reorder Buffer*) terminem, e escrevam

seus resultados nos respectivos registradores. Logo em seguida, os conteúdos do *Reorder Buffer* e do *Future File* são descartados. A espera das instruções precedentes ao desvio aumenta o tempo de latência médio dos desvios, entretanto, os experimentos do Johnson [JOHNS91] mostram que essa espera causa apenas uma redução de 4 a 5 % no desempenho geral do processador. Na maioria das vezes, o desvio é a instrução que fica na cabeça da fila do *Reorder Buffer* quando uma previsão errada é detectada. A razão disso é que as instruções de desvio apresentam normalmente dependências de dados em relação às instruções precedentes.

3.7 Tratamento de Interrupções Internas e Externas

Bem menos freqüentes que as previsões de desvio erradas, as interrupções internas e externas podem tolerar um tempo de resposta maior durante seu atendimento. Após uma interrupção o processador deve transmitir o estado em ordem consistente para as rotinas de tratamento de interrupção em *software*. Como todas as instruções estão sujeitas a algum tipo de exceção, o processador deve alocar todas as instruções no *Reorder Buffer*, mesmo que algumas delas não modifiquem o estado da máquina. Assim, a recuperação do estado consistente e reinício da execução funcionam da mesma forma como no caso de previsão de desvio incorreta.

Quando da ocorrência de interrupções externas, a solução é bastante simples. O processador pode verificar a condição de interrupção antes do despacho e decodificação das instruções. Se uma interrupção externa for detectada o processador simplesmente interrompe o despacho das instruções e espera que todas as instruções pendentes (despachadas mas não terminadas) terminem suas execuções. Após o término das instruções pendentes, o estado do processador será consistente.

Capítulo 4

Modelos de Simulação

Neste capítulo são descritos os modelos de máquina utilizados nos experimentos para avaliar o efeito da predição de desvios e interrupção precisa no desempenho de processadores Super Escalares. O modelo Super Escalar básico adotado, descrito em [FERNA92b], utiliza o algoritmo de despacho de instruções associativo (algoritmo de Tomasulo). Para viabilizar a execução especulativa, e com isso, melhorar o desempenho do modelo original, foram incorporados a esse modelo diversos mecanismos de predição de desvios e de interrupção precisa. Ao contrário dos modelos utilizados nos experimentos descritos em [FERNA92b], cujo objetivo era explorar ao máximo o paralelismo de instruções, estamos interessados em avaliar o efeito dos mecanismos de predição de desvios e de interrupção precisa num modelo Super Escalar realístico, com recursos semelhantes aos dos processadores Super Escalares atuais.

4.1 A Máquina Básica

Para efeito de comparação de desempenho, é ideal que a máquina Super Escalar proposta tenha a sua arquitetura básica derivada de uma máquina real, cujo desempenho se aproxime daquele permitido pelo estágio atual da tecnologia de implementação. Se a máquina básica escolhida apresentar um baixo desempenho, qualquer aplicação das técnicas Super Escalares seria ineficiente por causa das limitações já existentes na máquina. Ao contrário, se a máquina básica escolhida for uma máquina "ideal" (por exemplo, com um grande número de unidades funcionais com

latência curta), a aplicação das técnicas Super Escalares num modelo mais realístico teria um desempenho inferior ao da máquina "ideal", dificultando assim a avaliação do efeito das técnicas. Por esse motivo, decidimos adotar, como máquina básica, o processador i860 da Intel [INTEL89], que é um processador Super Escalar disponível no mercado. Os nossos modelos de simulação possuem as características básicas da arquitetura do processador i860, tais como o repertório de instruções, as memórias *cache* e os tipos de unidades funcionais.

Dois aspectos particulares da arquitetura do processador i860, o modo Dual (executa uma instrução de inteiro e uma de ponto flutuante simultaneamente) e as unidades funcionais *pipelined*, foram excluídos dos nossos modelos de simulação. As instruções do modo Dual do i860 dependem do escalonamento estático realizado pelo compilador, o que diverge da idéia de escalonamento dinâmico utilizado no modelo. No nosso ambiente de simulação, as instruções do modo Dual são simplesmente consideradas como instruções seqüênciais. Segundo os resultados dos experimentos realizados por Pleszkun [PLESZ88], a utilização de técnicas de *pipeline* nas unidades funcionais de um processador escalar não proporciona uma melhoria no desempenho superior a 4%. Por esse motivo e para simplificar o processo de modelagem da máquina simulada, as unidades funcionais do nosso modelo não utilizam a técnica *pipeline*.

Os bancos de registradores inteiros e de ponto flutuante do i860 foram mantidos no nosso medelo. As unidades funcionais do i860 também têm sua funcionalidade mantidas, com exceção da unidade *Core*. O freqüente conflito observado no uso da unidade *Core* pelas instruções de *Load/Store*, pelas instruções de aritmética e lógica inteira e pelas instruções de desvio, motivou-nos a criar uma outra unidade funcional de *Load/Store*, que executa exclusivamente as instruções de acesso à memória. A nova unidade *Core* somente fica responsável pela execução das instruções de aritmética e lógica inteira e de desvio.

Os tipos de unidades funcionais utilizados no nosso modelo e as suas respectivas funções são descritos a seguir:

Core - Responsável pela execução de todas as operações lógicas e aritméticas com

inteiros, instruções de desvio e de transferência de dados de um registrador inteiro para um outro de ponto flutuante.

Soma - Esta unidade executa as instruções de soma, subtração e comparação com reais, bem como a instrução de conversão de formato de ponto flutuante para inteiro.

Multiplicação - A unidade de multiplicação realiza as operações de multiplicação com reais e a avaliação do recíproco de um real (usada para implementar a divisão de reais).

Gráfica - Ela executa operações aritméticas sobre valores do tipo inteiro longo armazenados no banco de registradores de ponto flutuante. A unidade também inclui instruções usadas na implementação de algoritmos gráficos tridimensionias. Dentre estes algoritmos encontram-se o de sombreamento de *pixels* e o *Z-buffer* (para eliminar superfícies encobertas). A operação de transferência de dados dos registradores de ponto flutuante para os inteiros também é realizada pela unidade Gráfica.

Load/Store - As instruções de transferência de dados entre o sistema de memória e os bancos de registradores, são executadas por este tipo de unidade funcional.

4.2 O Modelo Super Escalar Básico

O modelo Super Escalar Básico, descrito em [FERNA92b], utiliza um algoritmo de escalonamento de intruções que busca, decodifica e despacha múltiplas intruções simultaneamente (a partir de uma janela de instruções), assumindo que a memória *cache* de instruções consiga fornecer instruções a uma taxa necessária (embora as falhas na *cache* sejam levadas em conta naquele modelo de arquitetura Super Escalar). Um conjunto de *reservation stations* foi associado a cada tipo de unidade funcional, e para implementar o esquema de rotulação do algoritmo de Tomasulo [TOMAS67], campos de *tags* e bits de ocupação (*busy bits*) foram acrescentados aos bancos de registradores. Além disso, um certo número de *Common Data Bus* (CDB) foram

incluídos para propagar os resultados produzidos pelas unidades funcionais, para as *reservation stations* e os bancos de registradores.

A busca, decodificação e despacho das instruções para as *reservation stations* ocorrem na ordem seqüencial indicada pelo programa. Entretanto, o início da execução, ou seja, a transferência das instruções das *reservation stations* para as unidades funcionais, podem ocorrer fora de ordem. A execução de uma instrução é iniciada somente quando todos os operandos fonte estiverem disponíveis na *reservation station* e quando uma unidade funcional estiver desocupada. Se tivermos mais de uma instrução pronta (nas *reservation stations*) para ser executada, aquela que foi escalonada há mais tempo terá prioridade. Pelo algoritmo de despacho, uma instrução pronta numa *reservation station* pode ultrapassar uma outra escalonada há mais tempo (mas que esteja esperando seus operandos) sendo executada antecipadamente pela unidade funcional.

Toda vez que uma instrução termina, um pedido de acesso ao CDB (para propagar o resultado da operação), é feito. Caso duas ou mais instruções terminem no mesmo tempo, a contenção no acesso ao CDB é resolvido por um esquema de prioridade, onde a instrução que iniciou há mais tempo recebe a prioridade maior.

Foi incorporado ao modelo básico uma memória *cache* de dados com o mesmo tamanho e organização do i860. A memória principal ficou organizada em um único banco, e todo acesso à memória de dados é realizada através da unidade de *Load/Store*. Introduzimos diferentes tempos de latência de acessos à *cache* para os casos de acerto (*cache hit*) e falha (*cache miss*).

O modelo básico possui uma única unidade *Load/Store*, a qual estão associadas um certo número de *reservation stations*. Dessa forma, apenas uma instrução de acesso à memória pode ser executada a cada momento. As instruções de *load* e *store* são despachadas para as *reservation stations*, e ficam organizadas em uma fila pela ordem de chegada. Para melhorar o desempenho da unidade *Load/Store*, um esquema de remoção de ambigüidades [BARBO92] foi incluído. Com esse esquema, algumas instruções de *load* e *store* podem ser antecipadas, desde que as condições para manter o estado consistente da memória sejam verificadas. Caso

exista mais de uma instrução capaz de ser antecipada, será dada prioridade àquela despachada há mais tempo.

Neste modelo, não foi utilizado nenhum método realístico de predição de desvios condicionais. Apenas dois esquemas fixos de tratamento de desvios condicionais foram adotados. O primeiro faz a previsão de desvios aleatoriamente, ou seja, ao decodificar uma instrução de desvio condicional, a chance do processador acertar o alvo do desvio é determinada aleatoriamente. O segundo esquema implementa um preditor onisciente, que prevê os desvios com 100 % de acerto. Esse último esquema é particularmente conveniente quando se deseja determinar o limite máximo teórico para o desempenho do modelo, pois ele elimina o efeito negativo das previsões erradas.

Os tempos de latência de cada unidade funcional são expressos em termos de ciclos de máquina. A Tabela 4.1 mostra as latências usadas no modelo básico.

Atividades	**Latências**
Decodificação de instruções	1 ciclo
Despacho de instruções	1 ciclo
Busca de instruções: *cache hit* *cache miss*	 1 ciclo 16 ciclos
Acesso à *cache* de dados: *cache hit* *cache miss*	 3 ciclos 18 ciclos
Operações de adição de ponto flutuante	6 ciclos
Operações de multiplicação de ponto flutuante: precisão simples precisão dupla	 6 ciclos 9 ciclos
Operações na unidade *Core*	3 ciclos
Operações na unidade gráfica	6 ciclos

Tabela 4.1: Tempos de Latência Usados no Modelo Super Escalar Básico.

A decodificação de instruções na simulação é dividido em duas fases:

- decodificação;
- busca de operandos

Na primeira fase, a decodificação das instruções propriamente dita é levada a cabo, e paralelamente, o resultado de uma unidade funcional é propagado pelo CDB, atualizando um registrador. Na segunda fase, a de busca de operandos, as instruções recebem os operandos requeridos (podem ser valores ou rótulos) a partir dos registradores (atualizados na primeira fase).

O escalonamento de instruções também foi dividido em duas etapas:

- despacho;
- avaliação do *CDB*

Durante o despacho de instruções, os parâmetros para a execução das instruções (obtidos durante a decodificação) são transferidos para as *reservation stations* livres. Na segunda etapa, as *reservation stations* verificam se o CDB está propagando algum resultado que será utilizado como operando. Caso afirmativo, a *reservation station* lê o operando do CDB, armazenando-o no campo apropriado. Através da monitoração do CDB após o despacho para as *reservation stations*, evita que uma instrução despachada fique esperando eternamente pelos operandos já propagados pelo CDB. Uma vez que todos os operandos e a unidade funcional correspondente estiverem disponíveis, a execução da instrução é iniciada.

Na simulação do modelo básico, quatro situações podem interromper o processo de escalonamento de instruções:

- quando a janela de instruções estiver vazia;
- quando não há *reservation station* disponível;
- quando ocorre uma previsão errada de desvio condicional;
- quando ocorre uma falha na *cache* durante a busca de instruções.

4.3 Modificando o Modelo Super Escalar Básico

O modelo modificado descrito nessa seção foi usado para avaliar o efeito da predição de desvios e da interrupção precisa em processadores Super Escalares. Esse modelo é derivado do modelo Super Escalar básico apresentado na seção anterior, e incorporou mecanismos de predição dinâmica de desvios e de interrupção precisa.

O modelo final adota os mesmos tempos de latência do modelo básico e o mesmo sistema de acesso à memória. Interessados em avaliar o efeito da previsão de desvios e da interrupção precisa num processador Super Escalar realístico e implementável, definimos o número de cada tipo de unidade funcional e o número de *reservation stations* associadas utilizando parcialmente a configuração balanceada obtida nos experimentos descritos em [FERNA92b]. Por exemplo, foram incluídas duas unidades *Core* conforme sugerido pela configuração balanceada. Tendo em vista que os números da configuração balanceada obtida em [FERNA92b] resultaram da execução de um conjunto limitado de programas de teste, procuramos utilizar também misturas de unidades funcionais utilizadas nos processadores Super Escalares encontrados no mercado. Os números de cada tipo de unidade funcional e de *reservation stations* estão listados na Tabela 4.2.

Tipo de Unidade Funcional	**Número de Unidade Funcional**	**Número de *reservation stations***
Core	2	8
Load/Store	1	6
Adição de ponto flutuante	1	2
Multiplicação de ponto flutuante	1	2
Gráfica	1	2

Tabela 4.2: Número das Unidades Funcionais e das *Reservation Stations* Associadas.

A capacidade de armazenamento da janela de instruções é de 4 instruções, ou seja, até 4 instruções podem ser escalonadas simultaneamente. Esse número está compatível com a realidade dos processadores RISC Super Escalares atuais, pois qualquer tamanho maior que 4 aumentaria muito a complexidade do processador, tornando a sua implementação inviável no estágio atual da tecnologia. Com o tamanho da janela de instruções igual a 4, somente 1 CDB foi usado no modelo. Essa escolha foi baseada no estudo do efeito do número de CDBs e do tamanho da janela de instruções no desempenho da máquina, realizado em [FERNA92c]. O estudo mostrou que para janelas com tamanho 4, o uso de 2 ou mais CDBs não apresentou ganho maiores do que 6% no desempenho geral da máquina, o que justifica

o uso de apenas 1 CDB no modelo.

A previsão dinâmica de desvios no modelo final é implementada através do *Branch Target Buffer*, descrito no Capítulo 2. O tamanho e a organização (mapeamento direto ou associativo) do *Branch Target Buffer* são os principais parâmetros de simulação do esquema de previsão. Além disso, outros parâmetros, tais como o algoritmo de previsão, os algoritmos de alocação e de substituição do BTB, também foram incluídos para medir suas influências no desempenho do modelo. Para efeito de comparação, algumas técnicas de previsão estática, tais como o preditor onisciente e previsão do tipo "o desvio sempre ocorrerá" também foram implementadas.

Quando uma instrução de desvio condicional é buscada pelo processador, é feito um acesso simultâneo ao BTB na busca desse desvio, e baseado nos dados nele encontrados a previsão do desvio é realizada. O fluxo é dirigido para a instrução indicada pela previsão sem que nenhum ciclo de atraso seja intruduzido. Caso o desvio não esteja no BTB, uma estratégia de previsão fixa é usada no lugar da dinâmica. Após o término da execução do desvio, se o processador descobrir que a previsão feita anteriormente estava incorreta, o processo de escalonamento de instruções é imediatamente interrompido e a seqüência correta de intruções deve ser retomada. Nesse caso, a penalidade imposta pela previsão errada seria a interrupção do escalonamento de novas instruções durantes o tempo de latência requerida para a busca da instrução alvo correta na memória, além da perda do processamento especulativo realizado após o desvio.

O esquema de interrupções precisas implementado no modelo final é o *Future File*. Como a inclusão de *Future File* (réplica do banco de registradores) não acarreta nenhum aumento no *overhead* do processador, e portanto não influencía o desempenho do modelo, apenas a parte de *Reorder Buffer* foi modelada e simulada. O número de entradas do *Reorder Buffer* é um dos parâmetros da simulação. Toda vez que uma instrução é despachada para uma *reservation station*, ela também é alocada no final da fila do *Reorder Buffer*. Uma instrução somente é retirada do *Reorder Buffer* quando ela terminar e quando todas as instruções precedentes também terminarem. Se não houver nenhuma entrada livre no *Reorder Buffer*, o

escalonamento de novas instruções ficará interrompido até que alguma entrada da fila do *Reorder Buffer* se torne livre. Essa é a situação adicional que provoca uma interrupção no escalonamento de instruções no modelo modificado em relação ao modelo básico.

Capítulo 5

Método de Avaliação

Neste capítulo, faremos uma breve descrição do ambiente desenvolvido para realizar os nossos experimentos, que inclui as ferramentas implementadas, a metodologia de simulação e os programas de teste adotados. Também são apresentadas as máquinas usadas como referência para comparar o desempenho do nosso modelo Super Escalar.

5.1 A Máquina Referência

Para avaliar o desempenho do modelo Super Escalar proposto (i.e., com predição de desvios e interrupções precisas), comparamos os resultados de simulação com o desempenho dos outros modelos que serviam como referência. No nosso caso, adotamos dois modelos de processador como referência. O primeiro é o modelo de processador seqüencial, onde as instruções são despachadas e executadas seqüencialmente. Uma instrução é buscada somente quando a anterior terminar a execução. O segundo é o modelo de processador Super Escalar com despacho associativo descrito em [FERNA92b], porém sem predição dinâmica de desvio e sem interrupções precisas. A comparação com o segundo modelo é de especial relevância, pois permite mostrar exatamente os efeitos da predição dinâminca de desvios e de interrupções precisas nos processadores Super Escalares com algoritmo de despacho dinâmico, que é o objetivo deste trabalho.

O modelo proposto e os dois modelos de referêcia possuem o mesmo número e os mesmos tipos de unidades funcionais. Os tempos de latência das

unidades funcionais também são iguais nos três modelos.

5.2 Programas de Teste (*Benchmarks*)

Para evitar avaliações errôneas do desempenho do nosso modelo Super Escalar, procuramos incluir, na bateria de programas de teste, componentes que melhor representem as diversas classes de aplicações. Utilizando esse critério, foram escolhidos 3 programas que fazem parte do pacote SPECint92 [SPEC92], que é muito usado na avaliação de desempenho das estações de trabalho RISC, além de 3 utilitários do sistema operacional UNIX e de programa *whetstone* (programa de teste sintético de ponto flutuante). A Tabela 5.1 mostra a descrição, o número de intruções simuladas e a percentagem de desvios (condicionais e incondicionais) desses programas de teste.

Nome	**Descrição**	**Instr.(10^3)**	**% Desvios**
espresso	Programa minimizador de expressões booleanas, para CAD em VLSI	827,7	22,6
compress	Compressor de arquivo usando codificação Lempel-Ziv adaptativa - comprimindo um arquivo texto de 40 KB	1.004,9	10,1
li	Interpretador de lingagem LISP	950,2	18,5
sed	Stream Editor - 1.13, utilitário do UNIX - editando um arquivo de 1,2 KB usando 28 regras	824,6	24,4
grep	grep-1.6, Busca em arquivo usando expressões regulares, utilitário do UNIX	1.001,9	19,5
diff	diff-1.16, utilitário do UNIX - comparando dois arquivos de $\approx$ 15 KB	929,8	22,6
whet	whetstone, benchmark sintético de ponto flutuante	1.025,6	9,3

Tabela 5.1: Descrição, Total de Instruções e Percentagem de Desvios dos Programas *Benchmark*.

Os programas **espresso**, **compress** e **li** pertencem ao pacote de *benchmark* de inteiros SPECint92. O programa **espresso** minimiza as funções de lógica booleana, e faz parte do pacote **Magic** de CAD para VLSI; o **compress** é

um compactador de arquivos usando codificação Lempel-Ziv adaptativa; e o **li** é um interpretador da linguagem LISP. Os utilitários do UNIX de uso freqüente que foram utilizados são o **sed**, o **grep** e o **diff**. *Sed* é um editor de texto não interativo que edita textos seguindo regras específicas; **Grep** realiza a busca de cadeias de caracteres dentro de um ou mais arquivos de texto; e **Diff** compara as diferenças entre dois arquivos de texto. O **whetstone** foi a única aplicação de ponto flutuante incluída nesse conjunto.

Todos programas de teste, escritos em liguagem C, foram traduzidos para o código objeto do processador i860 por um compilador C comercial, sem usar nenhuma otimização na compilação. Foi evitado, neste trabalho, o uso de programas de teste de poucas instruções com padrões de desvios bem definidos, tais como **Livermore Loops**, **Quick-Sort** e outros, pois esses programas poderiam levar a um resultado otimista na avaliação do esquema de predição de desvios. Todos os programas de teste possuem em média um milhão de instruções, e apresentam um fluxo de controle razoavelmente complexo. Devido à limitação dos recursos computacionais disponíveis e do tempo de simulaçao, não foi possível usar programas de teste com mais de um milhão de instruções. Entretanto, segundo Johnson [JOHNS91], o comportamento de programas de teste não muda muito após o primeiro milhão de instruções.

5.3 Técnica de Simulação

A técnica de simulação do tipo *Trace-Driven* foi usada na avaliação do desempenho do modelo Super Escalar proposto. A simulação *Trace-Driven* utiliza seqüências de instruções pré-determinadas, chamados *Traces*, que no nosso caso, são as seqüências de instruções dos programas de teste executadas pelo processador de referência seqüencial.

Os *traces* de instruções são gerados como mostra a Figura 5.1. O código objeto do programa de teste, gerado pelo compilador **pgcc** da Intel, é executado pelo simulador do processador i860 da Intel (**sim860**). Durante a execução, o simulador do i860 gera um arquivo de *trace* contendo várias informações acerca

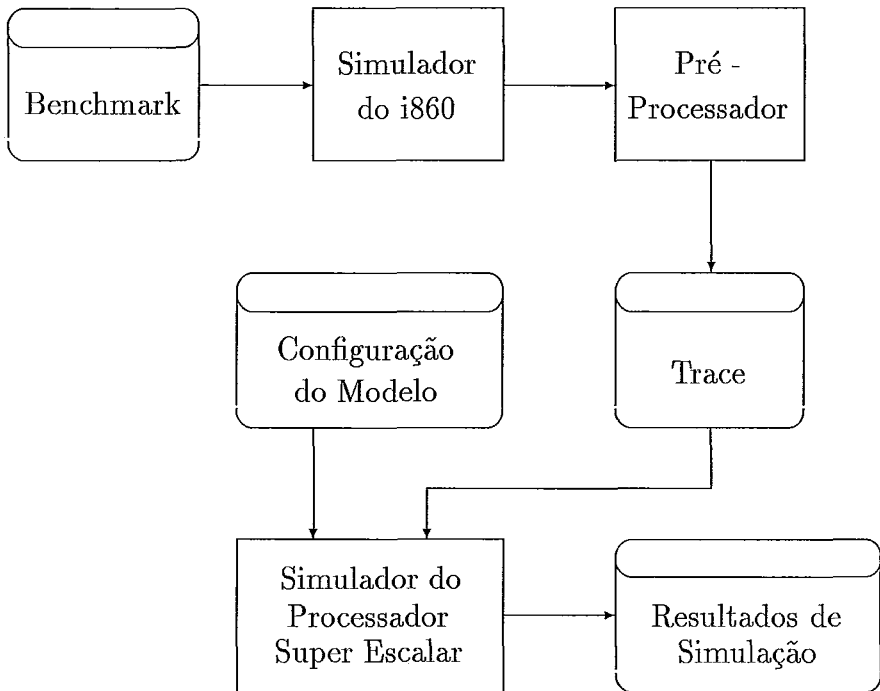


Figura 5.1: Diagrama de Fluxo da Simulação do Tipo *Trace- Driven*.

das instruções simuladas, tais como: o endereço da instrução, o código de operação, os registradores fonte e destino, e se houve uma falha na *cache* durante a busca da instrução. Em seguida, um pré-processador converte o *trace* gerado no formato reconhecido pelo simulador do nosso modelo Super Escalar. Além da conversão de formato, o pré-processador ainda filtra as instruções que não são processadas pelo simulador Super Escalar, como por exemplo, instruções do tipo *Pipelined* e *Dual Mode* do i860. As instruções do tipo *Dual Operation* do i860 (que executa uma operação de adição de ponto flutuante e uma de multiplicação de ponto flutuante simultaneamente) são transformadas em duas instruções de ponto flutuante seqüenciais (uma adição e uma multiplicação).

O simulador Super Escalar recebe, como entrada, o arquivo de *trace* convertido e um arquivo de configuração que determina a organização do processador Super Escalar simulado. Embora o simulador Super Escalar não execute efe-

tivamente as instruções , ele tenta realizar as mesmas operações de um processador Super Escalar real. O simulador busca as instruções do *trace* e contabiliza os tempos de atraso nas diversas fases de processamento das instruções, levando em conta as dependências entre as instruções, os conflitos no uso dos recursos, e as penalidades impostas pelos desvios.

O arquivo de configuração, que determina a organização do modelo Super Escalar, contém os seguinte parâmetros:

- Total de *reservation stations* associadas a cada tipo de unidade funcional
- Total de unidades funcionais de cada tipo
- Número máximo de instruções que podem ser escalonadas simultaneamente (Tamanho da Janela de Instruções)
- Total de CDBs
- Organização do *Branch Target Buffer* (mapeamento direto ou set-associativo)
- Número de entradas do *Branch Target Buffer*
- Algoritmo de predição do *Branch Target Buffer*
- Algoritmo de alocação do *Branch Target Buffer*
- Algoritmo de substituição do *Branch Target Buffer*
- Algoritmo de predição no caso de *miss* no *Branch Target Buffer*
- Tamanho do *Reorder Buffer*

Após a simulação de um programa de teste na máquina Super Escalar, é gerado um relatório contendo os resultados da simulação. Os principais resultados apresentados neste relatório incluem:

- Total de ciclos requeridos pelo programa na máquina Super Escalar
- Total de ciclos requeridos na máquina referência seqüencial

- Taxa de aceleração (*speed-up*)
- Taxa de redução
- Taxa de acerto na predição de desvios
- Taxa de *hit* e *miss* do *Branch Target Buffer*
- Percentagem do tempo de execução em que ocorreu bloqueio de despacho por conflito de recursos
- Percentagem do tempo de execução em que ocorreu bloqueio de despacho por penalidade dos desvios
- Percentagem do tempo de execução em que ocorreu bloqueio de despacho por falta de entrada do *Reorder Buffer*
- Taxa de ocupação do *Reorder Buffer*
- Número de instruções despachadas especulativamente
- Número de instruções executadas especulativamente
- Taxas de ocupação dos *reservation stations*, das unidades funcionais e dos CDB's

Neste trabalho, a taxa de aceleração (*speed-up*) obtida por uma máquina Super Escalar (para um certo programa de teste) é definida como a relação entre o número de ciclos requeridos pelo processador seqüencial para executar o programa e o número de ciclos requeridos pelo processador Super Escalar para executar o mesmo programa. A taxa de redução é o inverso da taxa de acelaração.

Uma série de simulações do modelo Super Escalar com diferentes parâmetros foi realizada. Com os resultados obtidos foi possível avaliar os efeitos da variação desses parâmetros no desempenho de máquinas Super Escalares.

Capítulo 6

Resultados de Simulação

Neste capítulo, serão apresentados os resultados da simulução do nosso modelo Super Escalar (descrito no Capítulo 4), que incorpora o algoritmo de despacho de múltiplas instruções, o *Branch Target Buffer* e o *Reorder Buffer* (com *Future File*).

Durante os nossos experimentos, variamos diversos parâmetros da arquitetura e estudamos seus efeitos no desempenho geral ou em alguns aspectos específicos do modelo Super Escalar. Os principais parâmetros avaliados incluem o tamanho e a organização do *Branch Target Buffer*, o algoritmo de predição, os algoritmos de substituição e de alocação utilizados no *Branch Target Buffer*, e o tamanho do *Reorder Buffer*.

Tendo em vista que estamos interessados em avaliar o impacto das alternativas de projeto num modelo de máquina Super Escalar realístico, utilizamos como principal critério de avaliação o compromisso "custo $\times$ desempenho".

Para visualizar melhor o efeito da variação de um certo parâmetro no modelo, procuramos fixar ou ajustar os demais parâmetros de modo que eles não interferissem na observação do efeito do parâmetro em estudo. Seguindo essa idéia, quando simulamos os mecanismos de predição de desvios usando o *Branch Target Buffer*, mantivemos para o modelo um *Reorder Buffer* com um número infinito de entradas, permitindo desse modo que o algoritmo de despacho de instruções não fosse interrompido por falta de entradas no *Reorder Buffer* (o que afetaria o desempenho da máquina, mascarando o efeito da predição de desvios). Analogamente, quando

estudamos o efeito do esquema de interrupções precisas usando *Reorder Buffer*, fixamos uma configuração do *Branch Target Buffer* com uma alta taxa de acerto nas previsões, e dessa forma, reduzimos ao mínimo a limitação imposta pelos desvios na exploração do potencial do modelo. Neste capítulo, para a simplificar a terminologia, quando falamos do "despacho" estamos nos referindo ao despacho de instruções para os *reservation stations* realizado pelo algoritmo de despacho associativo.

6.1 Impacto do Tamanho e da Organização do *Branch Target Buffer*

Para que o processador consiga executar as instruções especulativamente com eficiência, o mecanismo de predição de desvios deve fornecer uma elevada taxa de acerto. Além do algoritmo de predição e dos algoritmos de substituição e de alocação do BTB, a taxa de *hit* no BTB é um dos principais fatores que influenciam a taxa de acerto. Se o processador não conseguir encontrar no BTB uma entrada com a informação da previsão (uma falha no BTB), ele é obrigado a utilizar uma estratégia de previsão fixada, que pode aumentar bastante a chance de uma previsão errada. A taxa de *hit* no BTB depende fortemente da organização e do tamanho do BTB utilizado.

Durante nossos experimentos, empregamos 4 tipos de organizacão para o BTB: mapeamento direto, *2-way set-associative*, *4-way set-associative* e totalmente associativo. Para cada organização utilizamos BTBs com 2, 4, 8, 16,..., até 512 entradas, cada uma capaz de armazenar o endereço do desvio, o endereço do alvo e a informação de previsão de um desvio condicional. Nesses experimentos, utilizamos um *Reorder Buffer* com infinitas entradas, e adotamos a seguinte configuração para os demais parâmetros do BTB:

- algoritmo de predição dinâmica: 2 bits (vide Seção 2.3.3);
- algoritmo de alocação: *All-Branch* (todos os desvios são alocados, incluindo os tomados e os não tomados);
- algoritmo de subsituição: LRU;

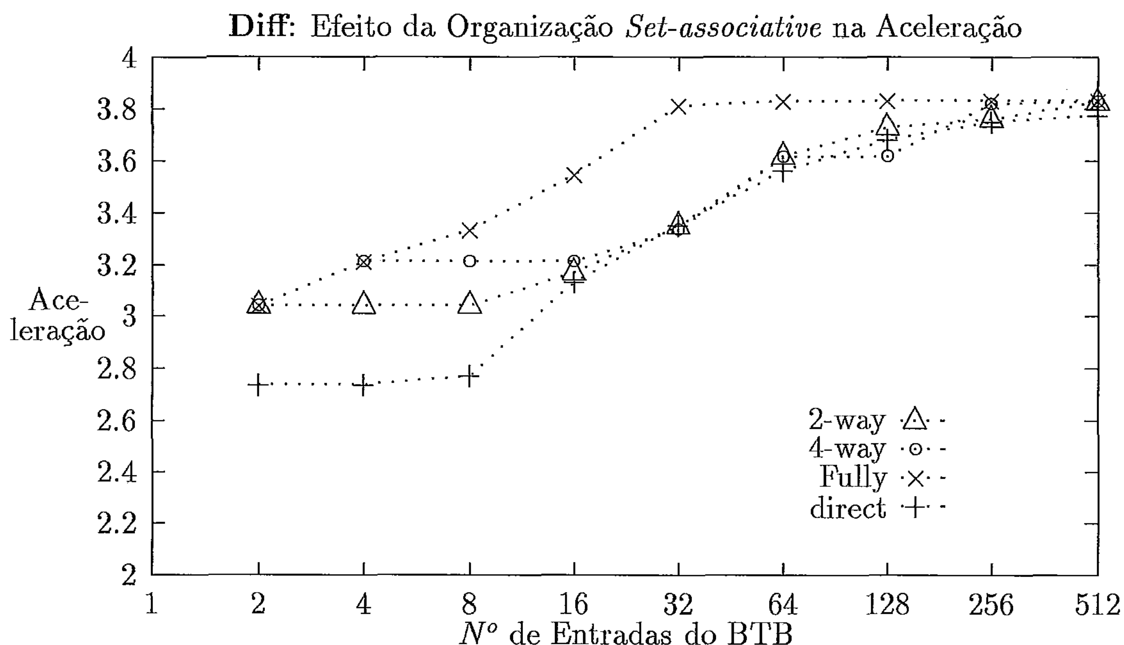


Figura 6.1: Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Diff).

- algoritmo de predição fixada (para caso de falhas no BTB): *Taken* (o desvio sempre ocorrerá).

A Figura 6.1 mostra o efeito na taxa de aceleração (*speedup ratio*) do programa **Diff** (em relação à máquina referência seqüencial), provocado pelo número de entradas do BTB e pela sua organização. Como podemos observar, a taxa de aceleração cresce com o tamanho do BTB. Entretanto, esse crescimento varia de acordo com o tipo de organização do BTB. Dentre as organizações simuladas, o BTB totalmente associativo (*fully associative*) apresentou a melhor taxa de aceleração. O BTB *2-way set-associative* e *4-way set-associative* apresentaram comportamentos semelhantes, exceto quando o número de entradas do BTB for menor que 16. Nesse caso, o *4-way set-associative* mostrou ser mais vantajoso do que o *2-way set-associative.* O BTB com mapeamento direto produziu as menores taxas de aceleração entre as 4 organizações, principalmente para BTBs com poucas entradas (inferiores a 32 entradas). Isso se deve à freqüente ocorrência de conflitos no

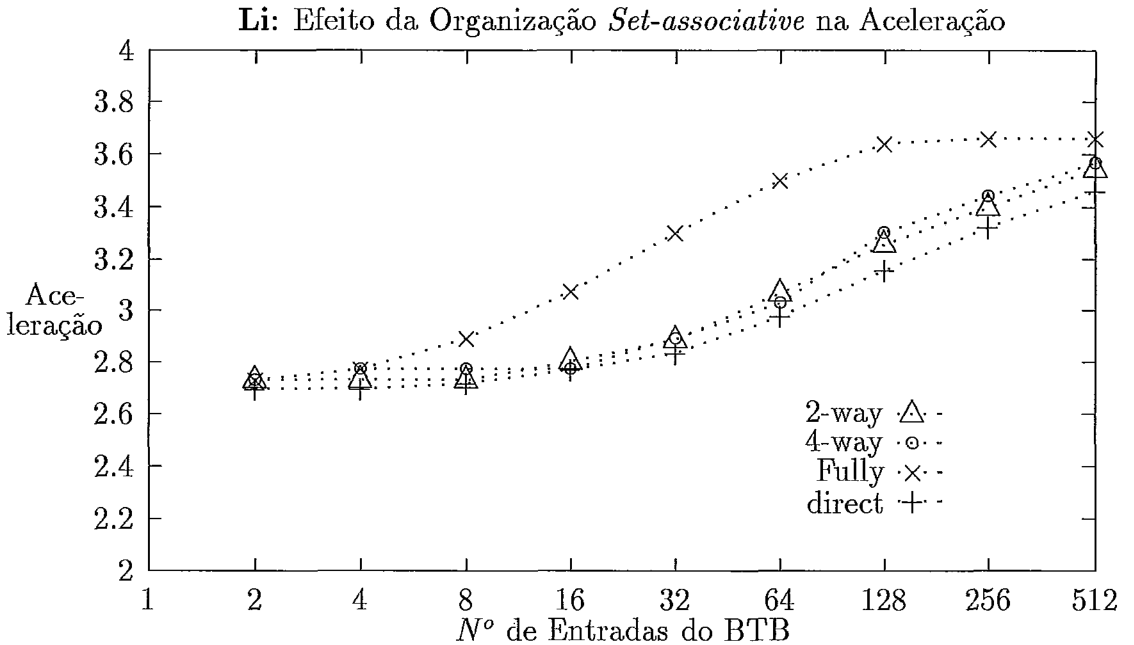


Figura 6.2: Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Li).

mapeamento dos endereços das instruções de desvio (resultam numa mesma entrada do BTB), aumentando consideravelmente a taxa de *miss* no acesso ao BTB (entre cerca de 40 a 80%).

Na organização totalmente associativa, os experimentos indicaram a ocorrência de um aumento acentuado na taxa de aceleração para BTBs com 2 até 32 entradas, porém a taxa estabilizou para BTBs com mais de 32 entradas. A taxa máxima varia de acordo com o programa de teste, alguns atingem valores próximos do máximo a partir de 16 entradas, enquanto outros a partir de 64 ou de 128 entradas. Essa variação depende do número e da distribuição dos desvios condicionais nos programas de teste. Todos os programas de teste apresentaram taxas de aceleração com características semelhantes - a organização totalmente associativo foi a que registrou a melhor taxa de aceleração para um mesmo número de entradas do BTB. As Figuras 6.2 e 6.3 mostram as taxas de aceleração dos programas **Li** e **Whetstone**. O programa **Li** apresentou um aumento suave na taxa de

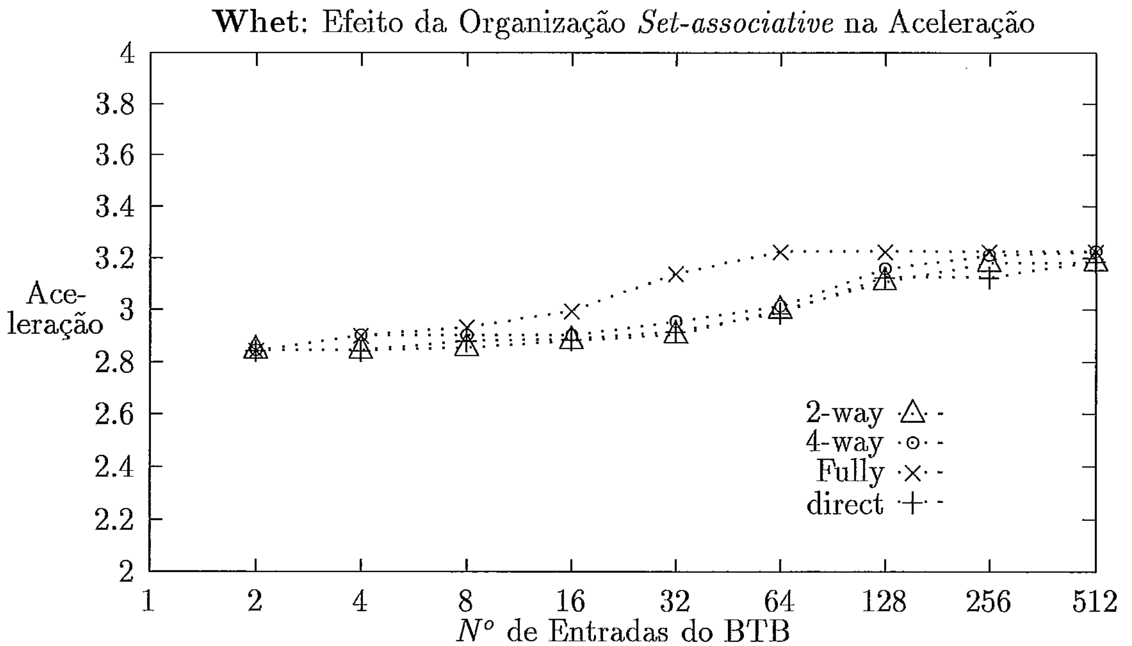


Figura 6.3: Efeito do Tamanho e da Organização do BTB na Taxa de Aceleração (Whetstone).

aceleração à medida que o número de entradas do BTB aumenta, com uma visível vantagem da organização totalmente associativa em relação às outras organizações. Já no programa **Whetstone**, que faz uso ostensivo das unidades de aritmética de ponto flutuante, registrou-se pouco ganho na taxa de aceleração apesar do aumento do número de entradas do BTB. Isso ocorre porque os conflitos no uso de recursos (decorrentes da alta taxa de utilização das unidades funcionais de ponto flutuante) limitaram o desempenho conseguido através da previsão de desvios.

A taxa de aceleração é função direta da taxa de acerto nas previsões de desvios que, por sua vez, depende de uma série de fatores do mecanismo de previsão. Quanto maior for a taxa de acerto, menor será a penalidade causada pelas previsões erradas, e maior aceleração será atingida. Comparando os valores da Figura 6.1 com os da Figura 6.4, obtidos do programa **Diff**, podemos verificar essa relação direta entre a taxa de aceleração e a taxa de acerto. Para o caso da organização totalmente associativa, taxas de acerto de até 98 ou 99% podem ser

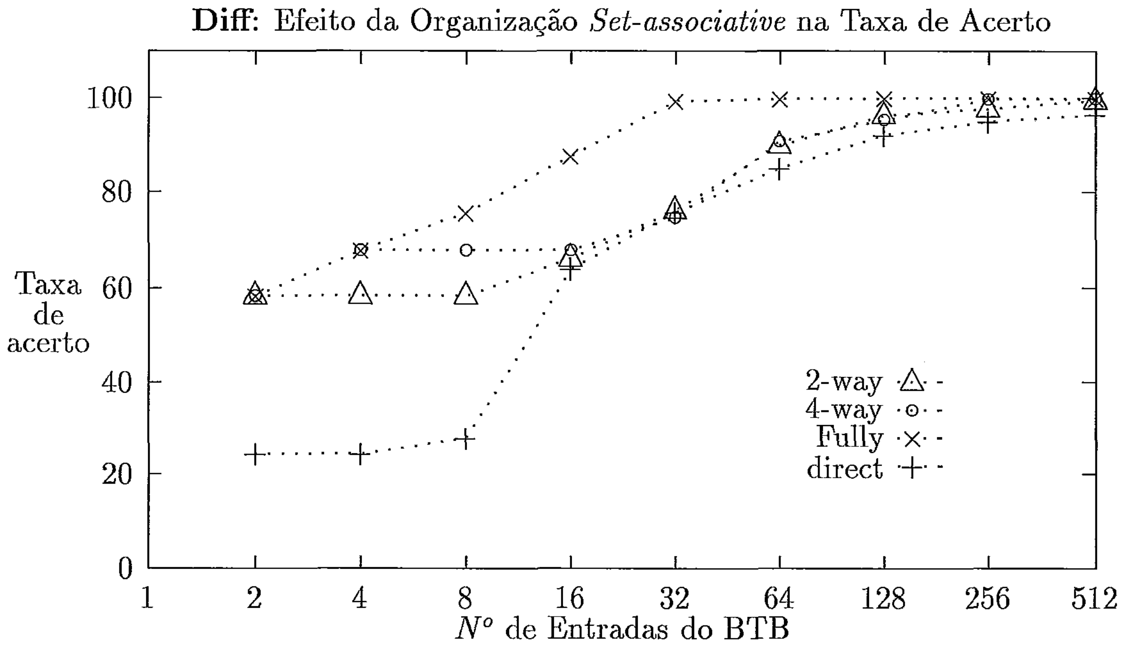


Figura 6.4: Efeito do Tamanho e da Organização do BTB na Taxa de Acerto (Diff).

alcançadas para BTBs com mais de 128 entradas. Entretanto, considerando o custo de implementação, o ganho relativo na taxa de acerto é pequena quando dobramos o número de entradas dos BTBs que possuem mais de 32 entradas.

Por outro lado, a taxa de acerto depende da taxa de *hit* do BTB. A Figura 6.5 mostra as curvas da taxa de *hit* do BTB (do programa **Diff**). Elas apresentam o mesmo perfil das curvas da taxa de acerto. Foi constatado que, na maioria dos casos, a taxa de acerto é ligeiramente maior que a taxa de *hit* (por exemplo, executando o programa **Grep**, o a organização totalmente associativa com 64 entradas registrou uma taxa de *hit* de 93,62%, enquanto a taxa de acerto foi de 93,77%). Isso é possível graças à estratégia de previsão fixada disponível no mecanismo. Quando a informação de predição de um desvio não estiver no BTB, o mecanismo ainda tem chance de prever corretamente o resultado do desvio utilizando a previsão fixada. A eficiência da previsão fixada depende do número de desvios condicionais tomados (ou não tomados) existentes nos programas. Obviamente, a previsão fixada apenas melhora a taxa de acerto no caso de uma falha no BTB. É

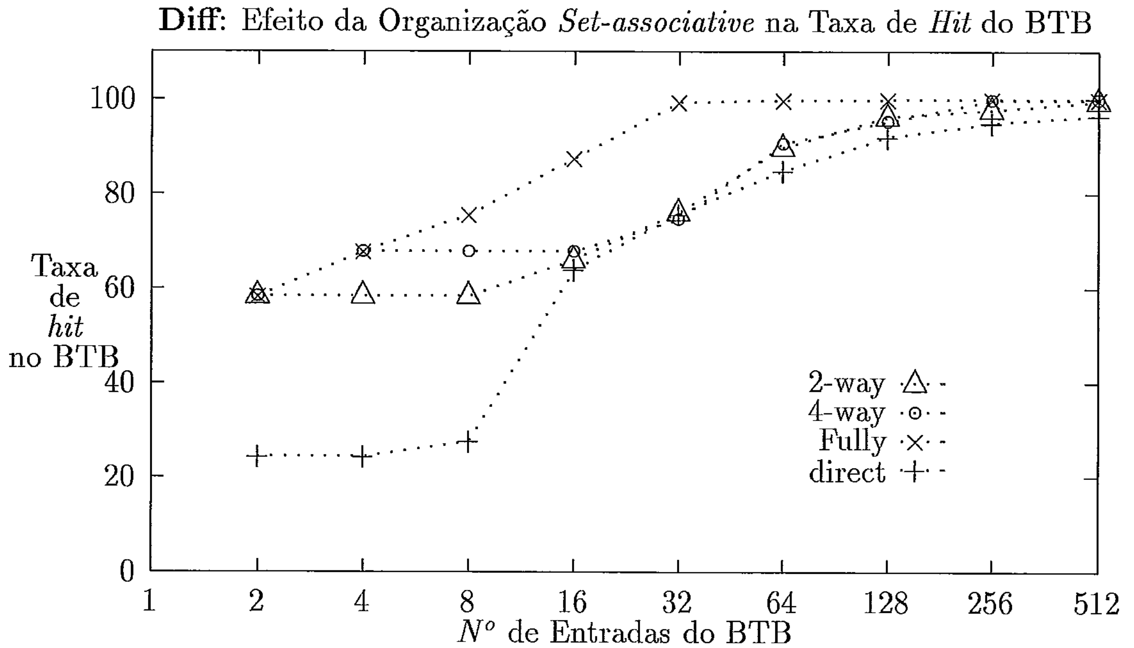


Figura 6.5: Efeito do Tamanho e da Organização do BTB na Taxa de *hit* do BTB (Diff).

sempre desejável tentar atingir a melhor taxa de *hit* possível.

Como foi visto anteriormente, o nosso modelo de processador interrompe o despacho de instruções toda vez que ele descobrir um erro na previsão de desvio, e somente reinicia o despacho quando o alvo correto é buscado. Nos experimentos, procuramos medir o tempo em que o processo de despacho ficou bloqueado por causa das previsões erradas. Quanto menor for a taxa de acerto, maior será o tempo total de bloqueio de despacho. Isso pode ser verificado na Figura 6.6, que mostra a percentagem do tempo de execução do programa **Li** em que o algoritmo de despacho permaneceu interrompido. No caso da organização totalmente associativa, essa percentagem atingiu 36,87 e 34,96% para BTBs respectivamente com 2 e 4 entradas, degradando o desempenho da máquina. Para BTBs com mais do que 128 entradas, o despacho de instruções ficou bloqueado em menos que 0,95% do tempo total de execução.

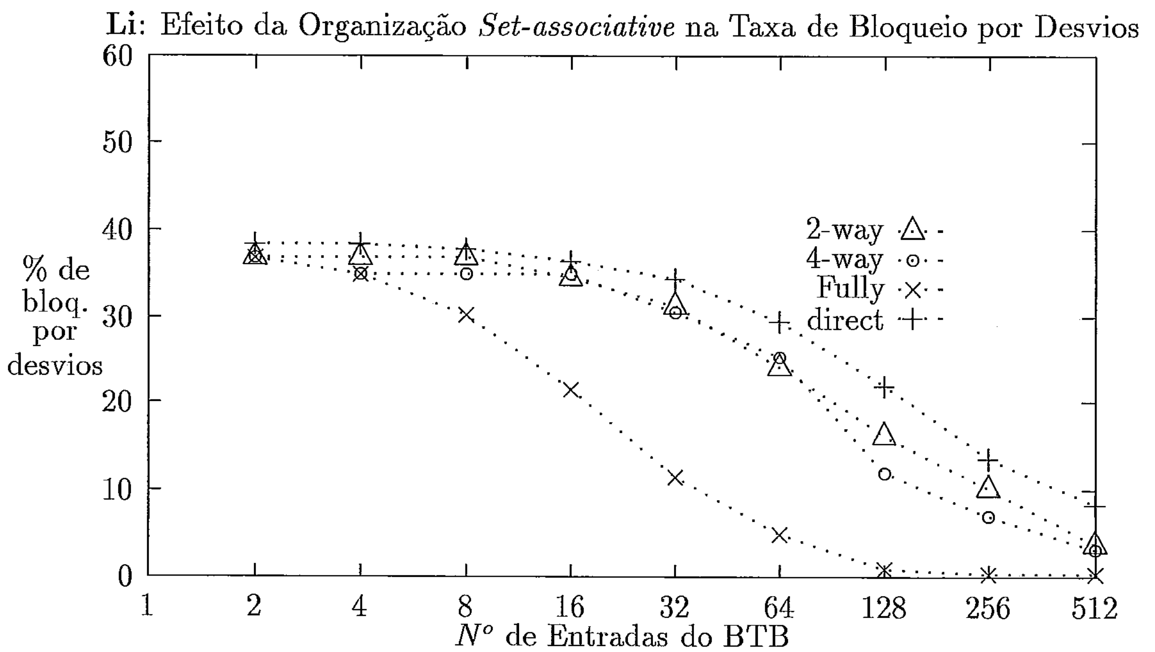


Figura 6.6: Percentagem do Tempo de Execução em que Ocorreu o Bloqueio de Despacho de Instruções por Desvios (Li).

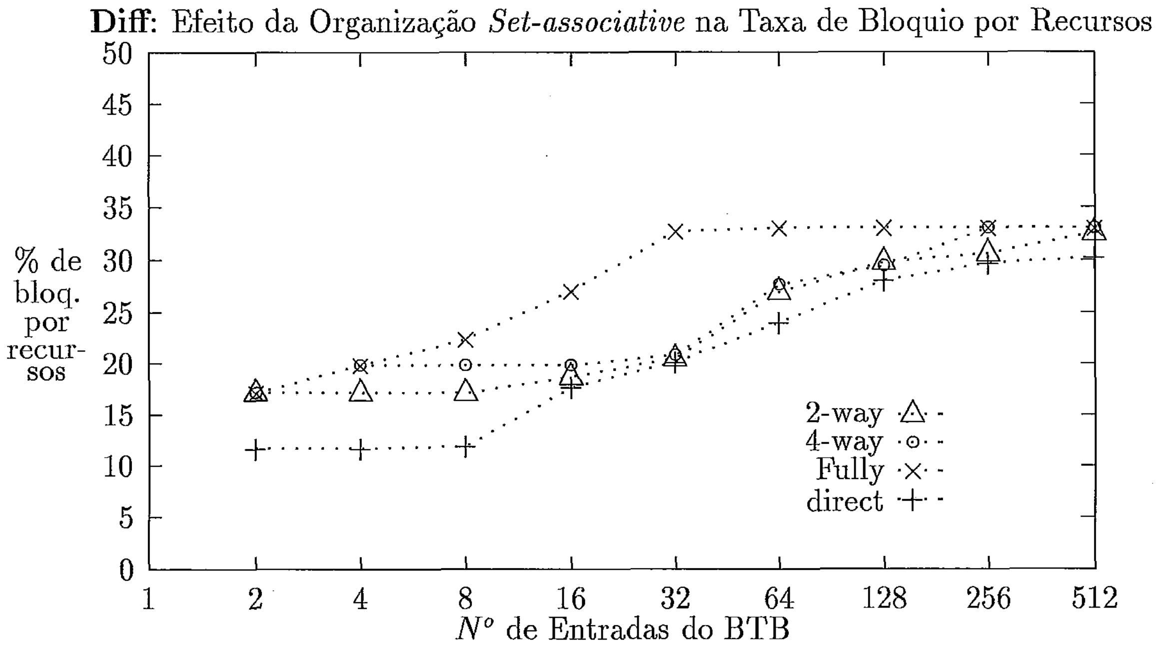


Figura 6.7: Percentagem do Tempo de Execução em que Ocorreu o Bloqueio de Despacho de Instruções por Falta de Recursos (Diff).

Nos programas de teste com um número maior de desvios condicionais (e.g., **Espresso**, **Sed** e **Grep**), verificamos uma percentagem maior na interrupção do algoritmo de despacho de instruções para BTBs pequenos (com um BTB de 2 entradas e mapeamento direto, taxas de bloqueio de 54,18%, 53,90% e 57,80% para **Espresso**, **Sed** e **Grep**, foram verificados). Ao contrário, o programa **Whetstone**, que possui poucos desvios condicionais, apresentou uma percentagem menor quando comparado com os demais programas (20,28% para um BTB com as características acima citadas). Em termos gerais, para todos os programas de teste, quando conseguimos alcançar uma taxa de acerto superior a 95%, a percentagem do tempo de execução em que ocorreu bloqueio de despacho por desvios ficou abaixo de 5%.

Quando a taxa de ociosidade do algoritmo de despacho é reduzida (por penalidade de desvios), aumenta o número de instruções despachadas por ciclo. É de se esperar que esse aumento provoque uma elevação na taxa de ocupação das

reservations stations associadas às unidades funcionais, até o ponto em que o despacho de instruções seja interrompido pela falta de recursos (i.e., *reservation station* livre). Como pode ser visto na Figura 6.7, que mostra a variação da percentagem do tempo de execução em que ocorreu bloqueio de despacho por falta de recursos, as curvas têm um aspecto inverso às da Figura 6.6. A ocorrência de bloqueios torna-se mais freqüente à medida que o tamanho do BTB é aumentado. Na Figura 6.7, a taxa de ociosidade do algoritmo de despacho (por falta de recursos) atingiu o patamar máximo (aproximadamente 33%) para BTBs com 32 entradas. Nesse ponto, a taxa de acerto também aproximou-se do seu valor máximo (cerca de 99,3%, vide Figura 6.4). Essa falta de recurso limita o desempenho da máquina, mantendo a taxa de aceleração em valores estáveis para BTBs com mais de 32 entradas (vide Figura 6.1). Taxas de aceleração maiores poderiam ser alcançadas se mais recursos (como *reservation stations* ou unidades funcionais) fossem incluídos.

Examinando os dados dos experimentos para avaliar o efeito do tamanho e da organização do BTB, podemos concluir que o BTB totalmente associativo fornece, sem dúvida, o melhor desempenho. Entretanto, o custo de implementação de um BTB totalmente associativo com muitas entradas (por exemplo, mais de 256 entradas) pode ser elevado demais para ser incluído num processador (no estágio atual da tecnologia). Nesse caso, o projetista poderia escolher a alternativa de um BTB *4-way set-associative*, ou mesmo um BTB com mapeamento direto, pois eles apresentam um desempenho bastante aproximado ao do BTB totalmente associativo com mais do que 256 entradas. Por outro lado, quando as limitações do projeto permitirem a inclusão de BTBs menores (até 32 entradas), é importante a utilização da organização totalmente associativa, pois ela apresenta um desempenho bastante superior às demais organizações.

6.2 Impacto do Algoritmo de Predição de Desvios

O algoritmo de predição utiliza informações sobre o que ocorreu anteriormente com um desvio (contidas no BTB) para prever o que ocorrerá durante a exeuc, ao corrente. Nos nossos experimentos, foram implementados dois algoritmos de predição

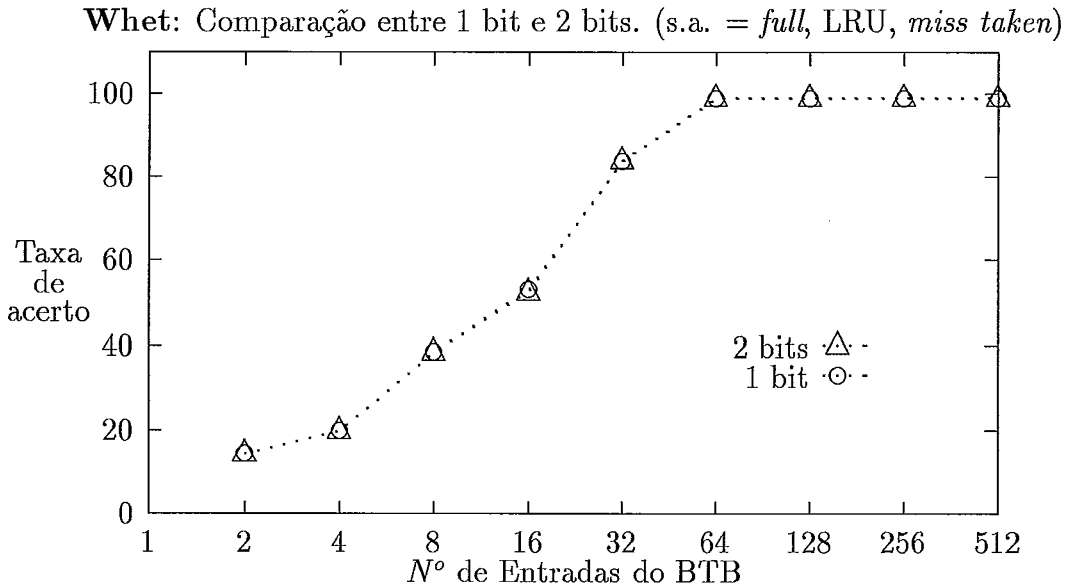


Figura 6.8: Comparação entre as Taxas de Acerto dos Algoritmos de Predição de 1 bit e de 2 bits (Whetstone).

de desvios: o algoritmo de 1 bit que utiliza o resultado do último desvio executado, e o algoritmo de 2 bits que utiliza o resultado dos dois últimos desvios executados. Os algoritmos foram descritos na Seção 2.3.3.

Para cada tipo de algoritmo, observamos o desempenho do modelo de máquina variando o tamanho do BTB. Os seguintes parâmetros do BTB foram mantidos iguais para os dois algoritmos de predição:

- organização do BTB: totalmente associativa;
- algoritmo de alocação do BTB: *All-Branch*;
- algoritmo de substituição do BTB: LRU;
- estratégia de previsão fixada: *Taken*.

Ao contrário do que se pensava antes dos experimentos, os algoritmos de predição de 1 bit e de 2 bits mostraram desempenhos semelhantes para todos os

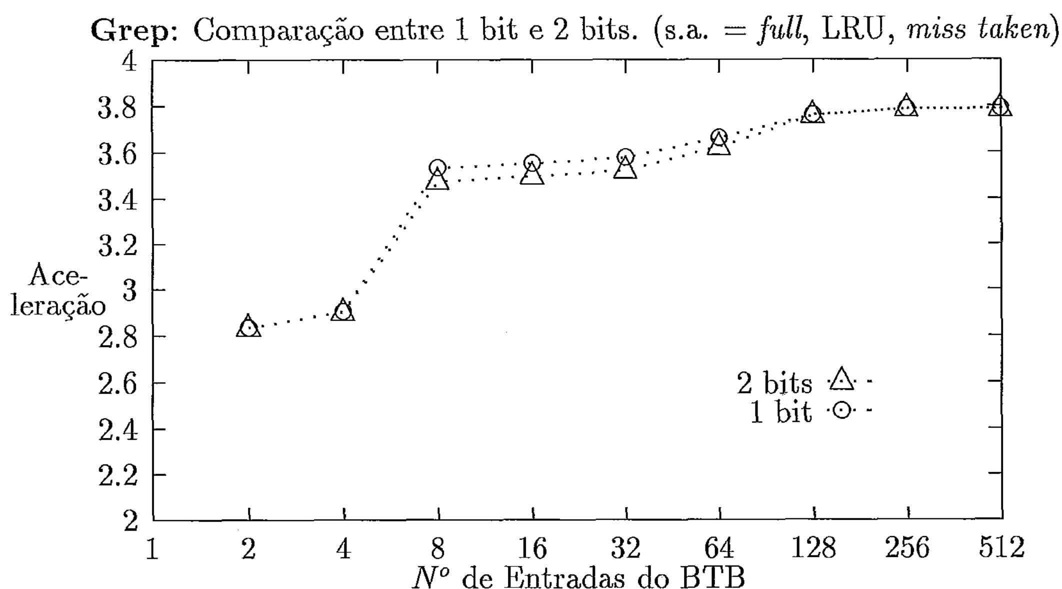


Figura 6.9: Comparação entre as Taxas de Aceleração dos Algoritmos de Predição de 1 bit e de 2 bits (Grep).

programas de testes utilizados neste trabalho. Isso pode ser verificado na Figura 6.8, que mostra a taxa de acerto nas previsões de desvios em função do tamanho do BTB, para o programa **Whetstone**. As curvas da taxa de acerto do algoritmo de 1 bit e do algoritmo de 2 bits estão praticamente sobrepostas, isso significa que, em termos numéricos, a diferença relativa no desempenho dos dois algoritmos é de aproximadamente 2,0%.

A Figura 6.9 mostra que as taxas de aceleração obtidas pelo algoritmo de 1 bit é ainda ligeiramente maior do que as obtidas pelo algoritmo de 2 bits (para o programa **Grep**). Examinando os resultados obtidos nas simulações e cosiderando o compromisso "custo × desempenho", podemos concluir que o uso do algoritmo de predição de 1 bit é mais vantajoso do que o de 2 bits, para programas que têm comportamentos semelhantes aos dos programas de teste usados. Aumentar a informação para a previsão contida no BTB de 1 bit para 2 bits significa dobrar a capacidade de armazenamento do campo de estatística, o que torna a implementação do BTB mais cara.

6.3 Impacto do Algoritmo de Substituição do *Branch Target Buffer*

Vimos anteriormente que a organização e o tamanho do BTB têm uma grande influência na taxa de *hit* do BTB, e dentre as organizações avaliadas, a *set-associative* foi a que apresentou o melhor desempenho. Nas organizações do tipo *set-associative*, a escolha do algoritmo de substituição (que remove o conteúdo de uma entrada no "*set*" quando um novo conteúdo é alocado), é de grande importância para o desempenho. Uma escolha inadequada do algoritmo de substituição pode causar a remoção de conteúdos freqüentemente acessados, gerando assim um grande número de falhas no BTB.

Um algoritmo de substituição amplamente utilizado, o LRU (*Least Recently Used*), remove o conteúdo da entrada menos recentemente acessada do conjunto. O algoritmo LRU exige que as entradas de cada conjunto do BTB sejam organizadas como uma fila ordenada pelo tempo de acesso. A última entrada da fila é a menos recentemente acessada, podendo ser removida quando um novo valor é alocado "*set*". Uma outra alternativa, de fácil implementação, é o algoritmo aleatório (*Random*). Esse algoritmo escolhe aleatoriamente a entrada que será removida dentro do "*set*". Realizamos experimentos que avaliaram o efeito desse dois algoritmos de substituição na taxa de *hit* e no desempenho global do modelo. Para cada algoritmo, variamos o tamanho do BTB. Os seguintes parâmetros do BTB foram mantidos iguais nos experimentos para comparar o desempenho dos dois algoritmos:

- organização do BTB: totalmente associativa;
- algoritmo de alocação do BTB: *All-Branch*;
- algoritmo de predição: 2 bits;
- estratégia de previsão fixada: *Taken*.

A Figura 6.10 mostra o resultado da comparação das taxas de acerto do BTB para o programa **Diff**, utilizando os dois algoritmos de substituição.

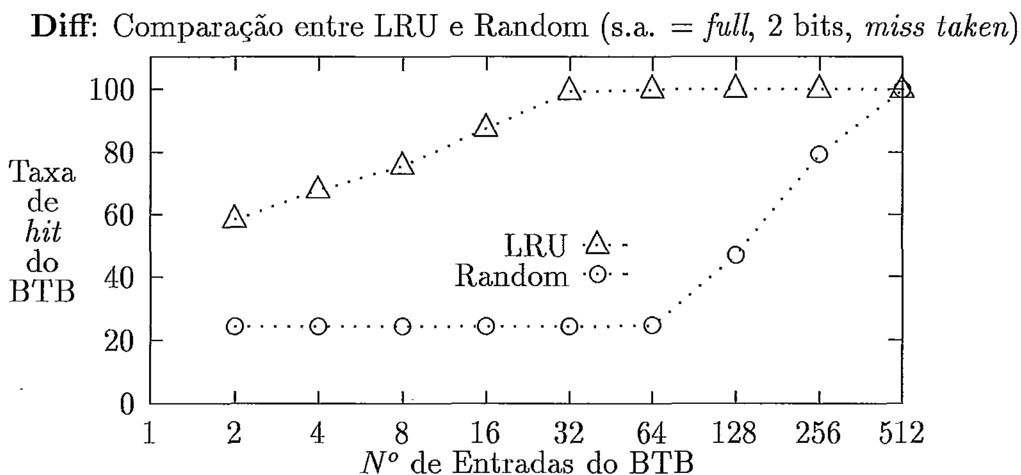


Figura 6.10: Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Diff).

Devido à localidade temporal das intruções de desvio nos programas, o algoritmo LRU apresentou uma visível vantagem em relação ao algoritmo aleatório. Com esse algoritmo, a taxa de acerto permaneceu em níveis bastante baixos (em torno de 24,5%) para BTBs com até 64 entradas. A partir de 64 entradas, a taxa de acerto cresceu gradativamente, até chegar 99,78% para 512 entradas, que é igual a taxa conseguida pelo algoritmo LRU.

Para o programa **Compress**, a taxa de acerto permaneceu em níveis baixos (aproximadamente 12,0%) para BTBs com até 128 entradas. Para BTBs com mais do que 128 entradas, um súbito aumento na taxa de acerto foi constatado. Esse aumento decorre do fato de que o BTB, a partir de um certo tamanho, passa a ser capaz de conter praticamente a grande maioria dos desvios mais freqüentemente executados do programa. Nessa circunstância, o algoritmo de substituição terá pouca influência na taxa de acerto.

Comportamento semelhante na taxa de acerto foi observado para os outros programas de teste, exceto para o programa **Whetstone**, onde os algoritmos LRU e *Random* apresentaram desempenhos semelhantes para todos os tamanhos do BTB (vide Figura 6.12). Uma das razões para essa semelhança é o número reduzido

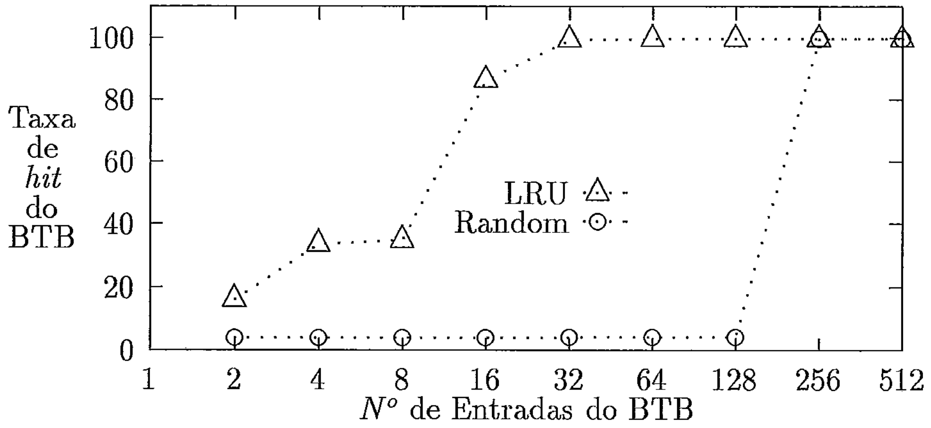


Figura 6.11: Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Compress).

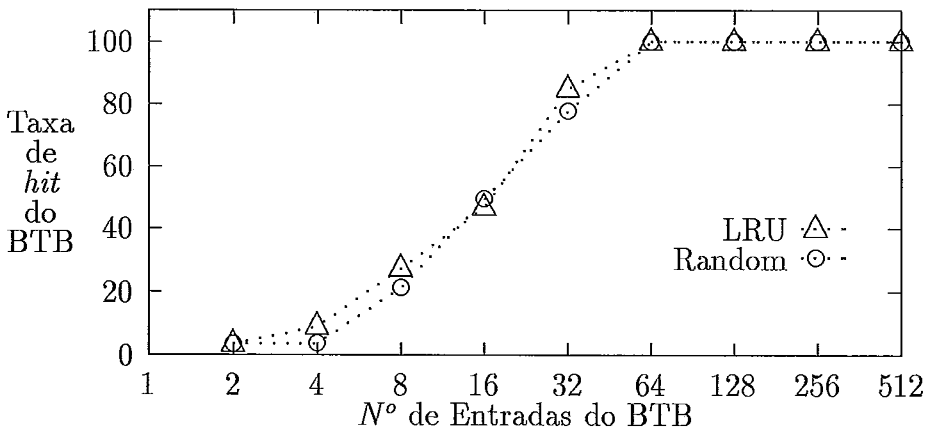


Figura 6.12: Comparação da Taxa de *Hit* entre os Algoritmos LRU e *Random* (Whetstone).

de desvios com endereços distintos encontrados no programa **Whetstone**. Diferente dos demais programas de teste, o **Whetstone** é caracterizado pela presença de *loops* regulares e com um grande número de iterações. Por esse motivo, o **Whetstone** possui poucos desvios com endereço distintos, o que o torna pouco sensível ao algoritmo de substituição para BTBs com tamanho pequeno.

Para um projetista, a escolha ideal do algoritmo de substituição dependerá da capacidade de armazenamento disponível para a implementação do BTB. Se o tamanho do BTB for limitado (por exemplo, menor do que 128 entradas), o algoritmo LRU será a melhor escolha. Para BTBs com mais de 256 entradas, o algoritmo aleatório, de fácil implementação, poderá ser utilizado pois ele não provoca maiores perdas no desempenho.

6.4 Impacto do Algoritmo de Alocação do *Branch Target Buffer*

Nos experimentos descritos nas seções anteriores, o algoritmo de alocação do tipo *All-Branch* foi usado. O algoritmo *All-Branch* aloca todos os desvios executados no BTB, sem levar em conta se o desvio foi tomado ou não. Na tentativa de utilizar melhor o espaço disponível (no caso de BTBs com tamanho reduzido), o algoritmo de alocação do tipo *Taken* surge com uma boa alternativa ao invés do algoritmo *All-Branch*. O Algoritmo *Taken* aloca no BTB apenas os desvios tomados. Considerando a hipótese de que o comportamento do desvio será o mesmo como ocorreu naúltima vez em que foi executado, não há necessidade de armazenar no BTB o endereço (ou instrução) alvo de um desvio não tomado, já que ele é o adjacente ao desvio.

A estratégia de previsão fixada (utilizada quando ocorre uma falha no BTB) deve ser escolhida de acordo com o algoritmo de alocação empregado. No caso do algoritmo de alocação *Taken*, é razoável prever que o desvio não será tomado se ele não estiver no BTB, pois não estando no BTB, significa que o desvio não foi tomado na última vez em que foi executado.

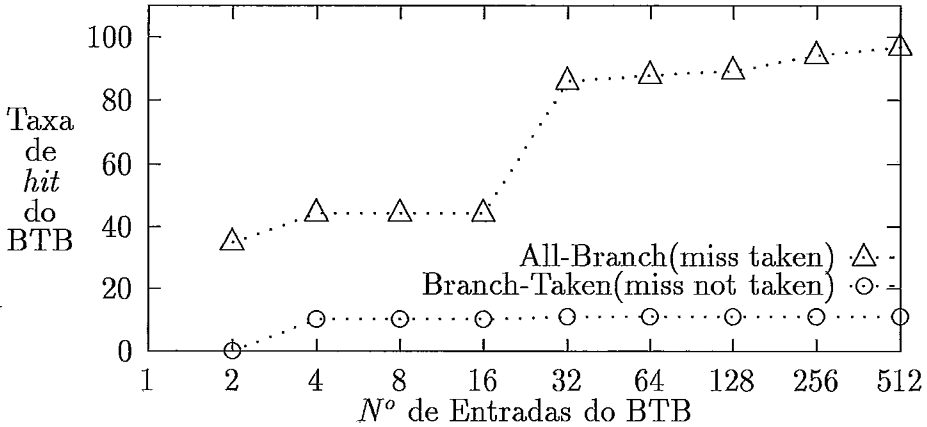


Figura 6.13: Comparação da Taxa de *Hit* entre os Algoritmos de Alocação *All-Branch* e *Taken* (Grep).

Podemos ver na Figura 6.13, a comparação das taxas de *hit* no BTB obtidas usando o algoritmo *All-Branch* (com a estratégia de previsão fixada "o desvio sempre ocorrerá") e o algoritmo *Taken* (com a estratégia de previsão fixada "o desvio nunca ocorrerá"). Foi observado que, enquanto a taxa de *hit* para o algoritmo *All-Branch* crescia com o tamanho do BTB, a taxa de *hit* para o algoritmo *Taken* se manteve sempre em níveis muito baixos (cerca de 10%). Embora esse resultado pareça estranho à primeira vista, existe um motivo para justificar esse baixo desempenho: de acordo com a estatística coletada nos *traces* de instruções utilizados, menos de 10% dos desvios condicionais foram tomados. Por esse motivo, apenas uma pequena parte dos desvios foram alocados no BTB, provocando freqüentes falhas no acesso ao BTB.

A existência de poucos desvios condicionais tomados nos *traces* dos programas de teste pode ser o resultado da otimização realizada pelo compilador do i860, usado durante a geração de *traces*. O compilador pode ter gerado o código de tal forma que as instruções sucessoras do desvio condicional (com maior probabilidade de execução) fossem colocadas nos endereços adjacentes ao do desvio, eliminando dessa forma o tempo de atraso requerido pela busca de instruç es fora da seqüência.

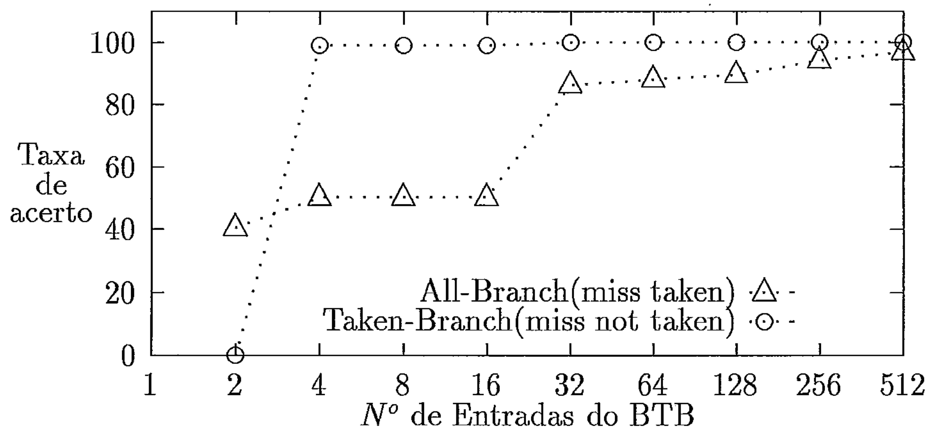


Figura 6.14: Comparação da Taxa de Acerto entre os Algoritmos de Alocação *All-Branch* e *Taken* (Grep).

A taxa de acerto obtida com o algoritmo *Taken* superou a taxa de acerto obtido com o algoritmo *All-Branch*, permanecendo acima de 99% independentemente do tamanhos do BTB. Essa alta taxa de acerto resulta da grande percentagem de desvios condicionais não tomados. A estratégia de previsão fixada "o desvio nunca ocorrerá" conseguiu prever praticamente todos os desvios não tomados que provocaram falhas no BTB.

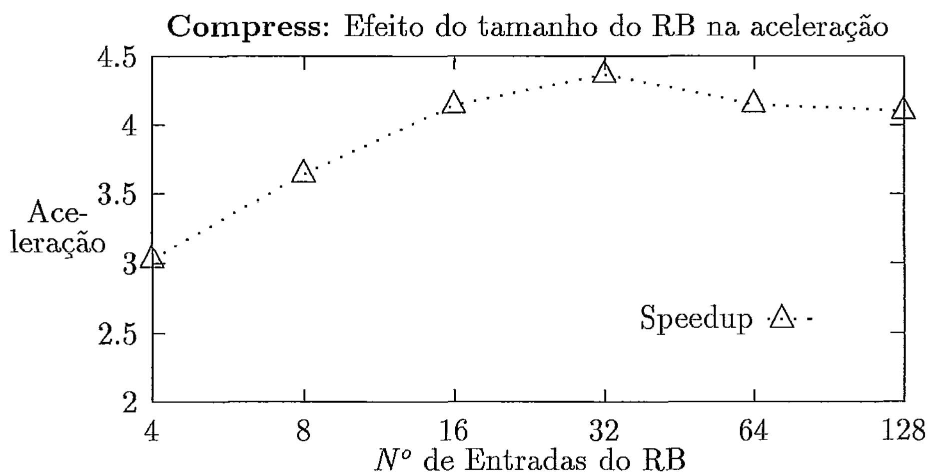


Figura 6.15: Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Compress).

6.5 Efeito do Tamanho do *Reorder Buffer*

O *Reorder Buffer* (RB), descrito na Seção 3.4, desempenha um papel fundamental no conceito de interrupções precisas. Dependendo da sua capacidade de armazenamento disponível, o *Reorder Buffer* pode impor um custo adicional no desempenho da máquina. Isso ocorre porque, no nosso modelo Super Escalar, o despacho de instruções é interrompido toda vez que o processador não encontrar entradas livres no RB. Portanto, o subdimensionamento do RB pode levar a sérias limitações no desempenho global da máquina. Por outro lado, aumentar indiscriminadamente o tamanho do RB não trará necessariamente ganhos no desempenho, mas sim um aumento no custo de implementação. Por essa razão, experimentos foram conduzidos para avaliar o efeito do tamanho do RB no desempenho da máquina, e com isso determinar um tamanho ideal para o modelo Super Escalar proposto.

Para evitar que o mecanismo de predição de desvios interfira na avaliação do efeito do RB no desempenho, utilizamos uma mesma configuração do *Branch Targer Buffer* que forneça uma alta taxa de acerto em todos os experimentos envolvendo o RB. Os parâmetros dessa configuração de BTB são:

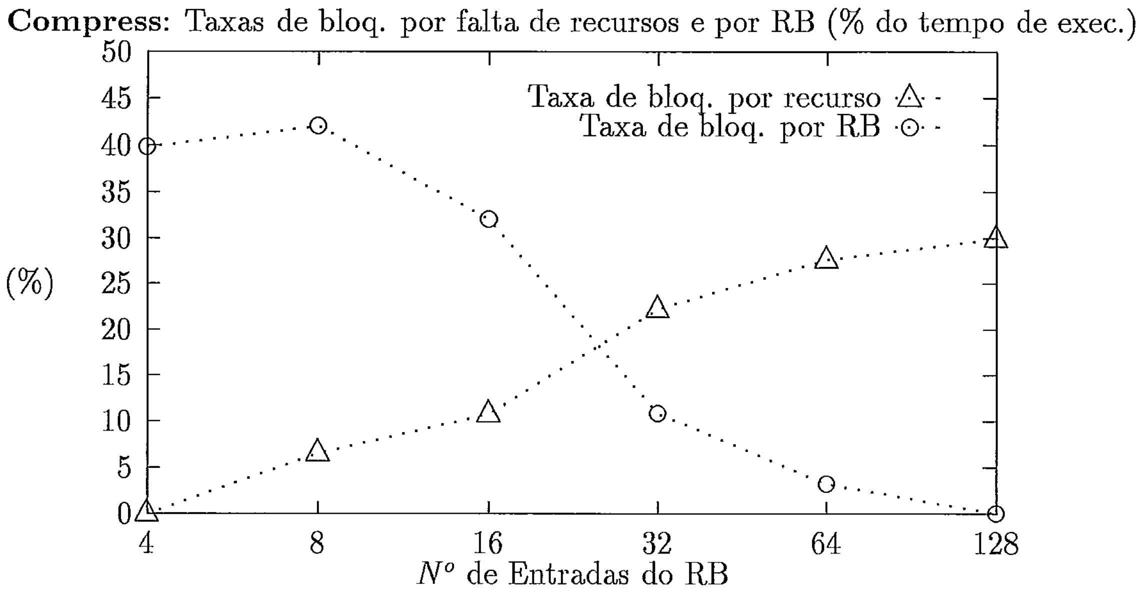


Figura 6.16: Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Compress).

- organização do BTB: totalmente associativa;
- tamanho do BTB: 512 entradas;
- algoritmo de previsão: 2 bits;
- algoritmo de alocação do BTB: *All-Branch*;
- algoritmo de substituição do BTB: LRU;
- estratégia de previsão fixada: *Taken*.

Nos experimentos, utilizamos RBs com 4, 8, 16, 32, 64 e 128 entradas. RBs com tamanho menor do que a janela de instruções (4 instruções) não foram incluídos nos experimentos, pois causariam um número excessivo de bloqueios no despacho das instruções. A Figura 6.15 mostra a variação na taxa de aceleração obtida pelo programa **Compress**, para diversos tamanhos do RB. Para RBs com 4 entradas, a taxa de aceleração ficou em 3,03. A partir de 4 entradas, a taxa

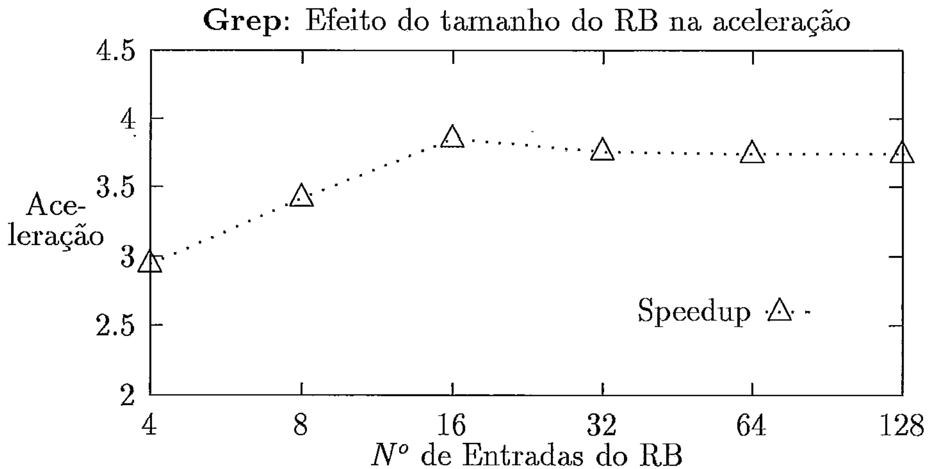


Figura 6.17: Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Grep).

de aceleração cresceu continuamente até atingir o máximo de 4,36 para RBs com 32 entradas, o que representou um aumento significativo (de 43,9%) na taxa de aceleração. Essa melhoria no desempenho foi possível graças ao aumento do tamanho do RB, reduzindo assim, a chance do algoritmo de despacho ficar bloqueado por falta de entradas no RB. A percentagem do tempo de execução em que houve bloqueio de despacho por falta de entradas no RB está mostrado na Figura 6.16. Para RBs com 4 entradas, durante 39,76% do tempo total de execução, o processador ficou esperando por uma entrada livre no RB. À medida que o tamanho do RB cresce, essa percentagem cai rapidamente, chegando a 0,02% para RBs com 128 entradas.

Embora o aumento do tamanho do RB reduza a taxa de ociosidade do do algoritmo de despacho de instruções, nem sempre taxas de aceleração maiores são atingidas. Por exemplo, conforme pode ser verificado na Figura 6.15, para RBs com entradas maiores que 32, a taxa de aceleração caiu de 4,36 para 4,10. O motivo dessa queda está no aumento da taxa de ociosidade do algoritmo de despacho por falta de recursos. A Figura 6.16 mostra que ocorreu um aumento contínuo na percentagem dessa taxa de ociosidade.

Na verdade, a taxa de aceleração depende do percentual do tempo

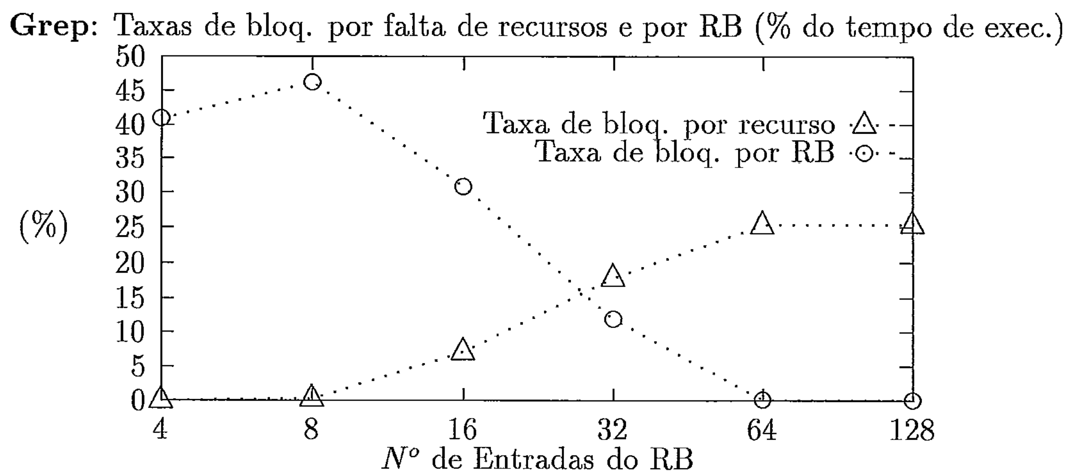


Figura 6.18: Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Grep).

de execução em que ocorreu bloqueio de despacho tanto por falta de recursos como por falta de entradas no RB. A taxa de aceleração máxima foi atingida para RBs com 32 entradas, exatamente quando o efeito composto da degradação causada por falta de recursos e por falta de entradas no RB, era mínimo (vide Figura 6.16).

Para o programa **Grep**, a taxa de aceleração atingiu o seu valor máximo para RBs com 16 entradas (vide Figura 6.17), pois a percentagem do tempo de execução em que ocorreu bloqueio de despacho por falta de recursos somente mostrou aumento expressivo para RBs com mais de 16 entradas (vide Figura 6.18). Como conseqüência da estabilização na taxa de ociosidade do despacho por falta de recursos, verificamos uma pequena queda na taxa de aceleração a partir de 16 entradas, que se estabilizou em torno de 3,74 para RBs com mais de 64 entradas.

Pela natureza do seu código, o programa **Whetstone** apresentou um tempo de bloqueio de despacho por falta de recursos bastante alto quando comparado com os desmais programas, inclusive para RBs pequenos (vide Figura 6.19). A percentagem do tempo de execução em que ocorreu esse bloqueio chegou a 37,72 e 44,13% para RBs com 16 e 64 entradas, respectivamente. Embora o tempo de

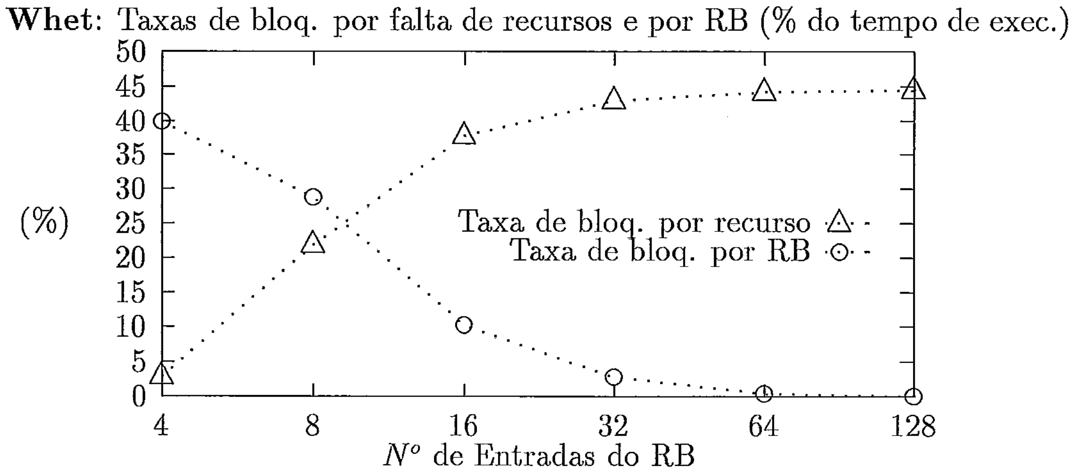


Figura 6.19: Efeito do Tamanho do *Reorder Buffer* no % do Tempo de Execução em que Ocorreu Bloqueio de Despacho de Instruções por Falta de Entradas no RB e por Falta de Recursos (Whetstone).

bloqueio de despacho por falta de entradas no RB caia rapidamente com o aumento do tamanho do RB, o crescimento na taxa de aceleração ficou limitado pelos conflitos no uso de recursos. O gráfico da Figura 6.20 mostra que a taxa de aceleração máxima conseguida pelo programa **Whetstone** foi de apenas 3,28.

Examinando os resultados dos experimentos anteriores, podemos ver que a degradação no desempenho causada pelo bloqueio no despacho por falta de entradas no RB, pode ser minimizada se utilizarmos RBs com poucas entradas. Para todos os programas de teste, a percentagem do tempo em que ocorreu esse tipo de bloqueio no despacho cai em média para 10% para RBs com apenas 32 entradas. O uso de RBs com mais de 32 entradas não trará benefícios significativos na diminuição deste tempo de bloqueio.

Comportamentos semelhantes foram observadas nas curvas da taxa de acertos e dos tempos de bloqueio de despacho por falta de recursos e por falta de entradas no RB, para os outros programas de teste. Após a análise desses comportamentos, podemos concluir que a escolha do tamanho ideal do RB dependerá da capacidade do despacho (número de instruções que a máquina consegue despachar

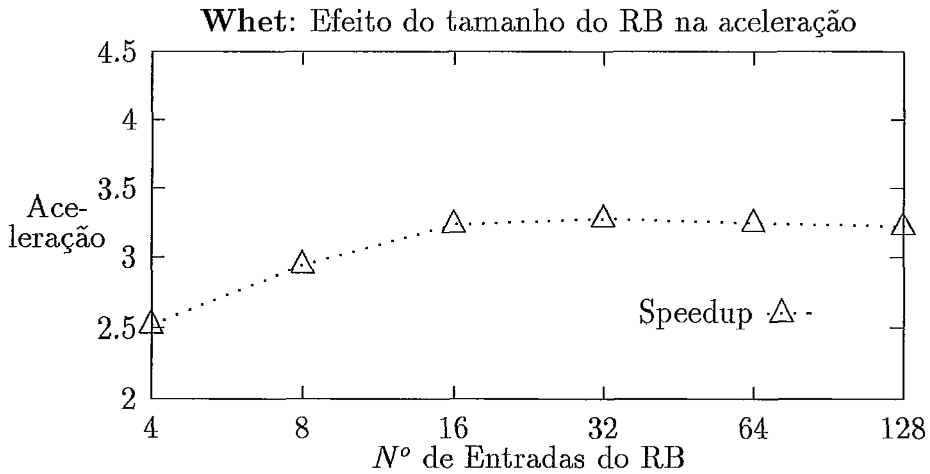


Figura 6.20: Efeito do Tamanho do *Reorder Buffer* na Taxa de Aceleração (Whetstone).

por ciclo) e do número de recursos disponíveis da máquina. Um RB muito pequeno pode limitar a taxa de despacho de instruções, subutilizando a capacidade do algoritmo de despacho. Por outro lado, aumentar simplesmente o tamanho do RB, sem levar em conta as limitações de recursos, elevará apenas o custo da máquina.

Para o tamanho da janela de instruções e a mistura de unidades funcionais utilizadas no nosso modelo Super Escalar, um RB com 32 entradas seria uma boa opção levando-se em conta o compromisso "custo × desempenho".

6.6 Capacidade de Execução Especulativa

Graças aos mecanismos de predição dinâmica e de interrupções precisas, o processador antecipa a execução das instruções pertencentes ao caminho mais provável do desvio, antes dele ser realmente executado. Essa capacidade de execução antecipada é chamada **Execução Especulativa**.

Num modelo de máquina Super Escalar, é importante conhecermos a capacidade de execução especulativa da máquina. Em outras palavras, devemos

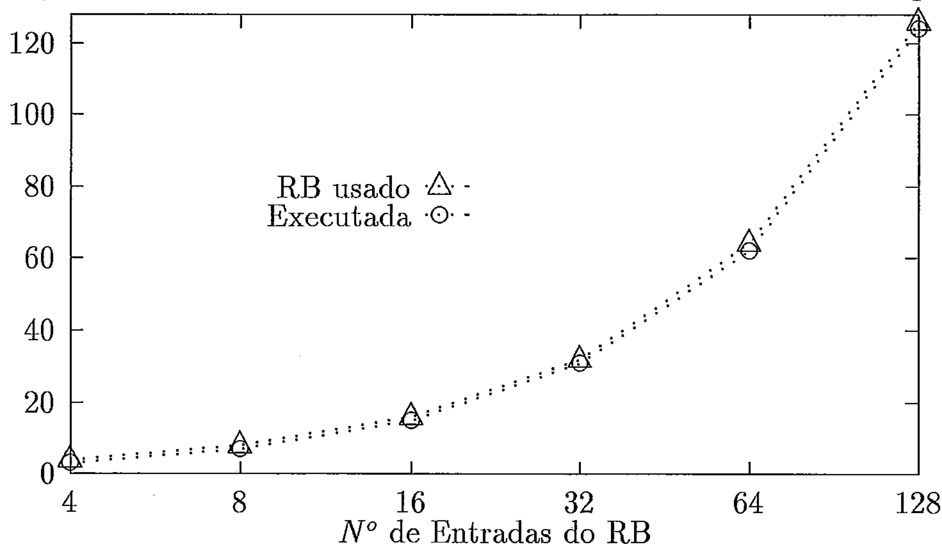


Figura 6.21: Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Compress).

avaliar características tais como, o número de instruções que a máquina é capaz de despachar especulativamente (i.e., antes que alguma instrução de desvio condicional precedente termine), e o número de instruções que já terminaram ou estão sendo executadas especulativamente. Esses valores podem ser facilmente obtidos examinando o conteúdo do RB, onde são armazenadas as instruções despachadas (podem estar terminadas ou não) na ordem seqüencial especificada pelo programa.

Consideramos como especulativas as instruções subseqüntes à primeira instrução de desvio condicional no RB que ainda não terminou. Dependendo do resultado do desvio, essas intruções podem ser executadas ou não. Essas instruções podem inclusive conter outras instruções de desvios condicionais, possibilitando assim a execução especulativa de vários níveis de desvios condicionais.

Para obtermos o número de instruções despachadas especulativamente pelo processador, basta contabilizar o número de instruções entre a primeira instrução de desvio condicional não terminada na fila do RB e o final da fila do RB.

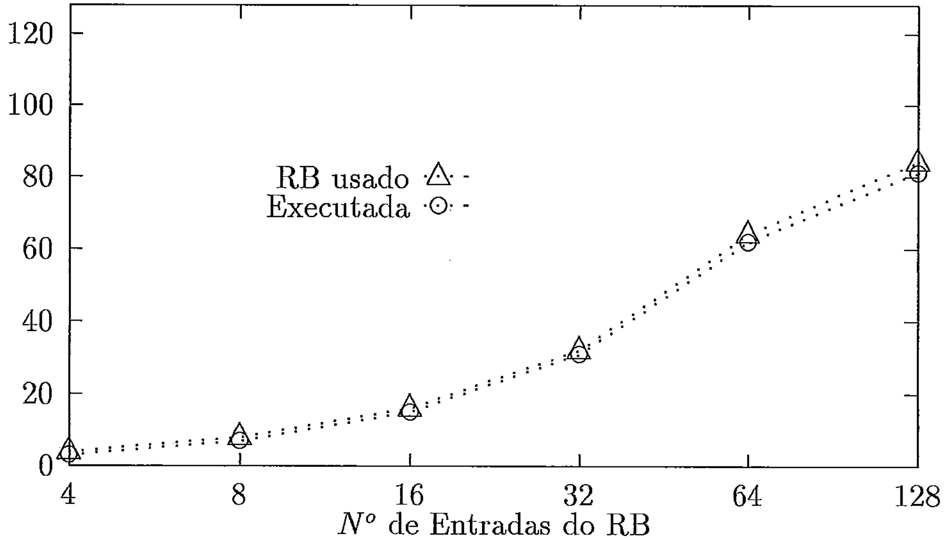


Figura 6.22: Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Espresso).

As instruções despachadas especulativamente ficam aramazenadas nas *reservation stations*, alguma delas podem estar sendo executadas pelas unidades funcionais e outras podem estar esperando por algum operando para serem iniciadas.

O número de instruções executadas especulativamente (inclui as já terminadas e as em fase de execução) é igual a diferença do número de intruções despachadas especulativamente e do número das instruções ainda não iniciadas (por causa de alguma dependência de dados ou conflito de recursos). Por esse motivo, é de se esperar que o número de instruções executadas especulativamente seja menor do que o das despachadas especulativamente.

Nos nossos experimentos, observamos as seguintes características da execução especulativa em função do tamanho do RB:

- o número médio de entradas ocupadas do RB;
- o número médio de instruções despachadas especulativamente;

- o número médio de instruções executadas especulativamente;
- o número máximo de entradas ocupadas do RB;
- o número máximo de instruções executadas especulativamente;
- os números máximo e médio de instruções de desvios despachadas especulativamente.

Antes de analisarmos o número de instruções despachadas e executadas especulativamente, vamos examinar primeiro a ocupação das entradas do RB. A Figura 6.21 mostra o número máximo de entradas do RB que foram ocupadas durante a execução do programa **Compress**, e o número máximo de instruções executadas especulativamente. Podemos ver na figura que, para todos os tamanhos do RB simulado, o processador chegou a utilizar, em certos momentos da execução, praticamente todas entradas disponíveis do RB. Mesmo para RBs com 128 entradas, até 126 entradas foram utilizadas em algum instante da execução. Entretanto, isso não significa que a taxa de ocupação do RB permaneceu nesse nível alto, durante toda execução. Como será visto mais adiante, a taxa média de ocupação do RB no tempo é bem inferior ao valor máximo. O número máximo de instruções executadas especulativamente, como pode ser visto na mesma figura, se aproxima muito do número máximo de entradas do RB ocupadas (124 instruções para RBs com 128 entradas).

Já para o programa **Espresso**, as entradas ocupadas não foram tantas como no programa **Compress**. Para RBs com 128 entradas, o número máximo de entradas ocupadas foi 84 (vide Figura 6.22), e para RBs menores do que 64 entradas, o processador também chegou a ocupar todas as entradas do RB. O programa de teste que apresentou a menor taxa de ocupação máxima foi o **Whetstone**. Durante a execução do **Whetstone**, o processador nunca ocupou mais do que 56 entradas do RB (vide Figura 6.23).

Os valores máximos do número de entradas do RB ocupadas e de instruções executadas especulativamente serviram para dar uma idéia dos limites máximos da execução especulativa do modelo. Entretanto, para avaliar a eficiência

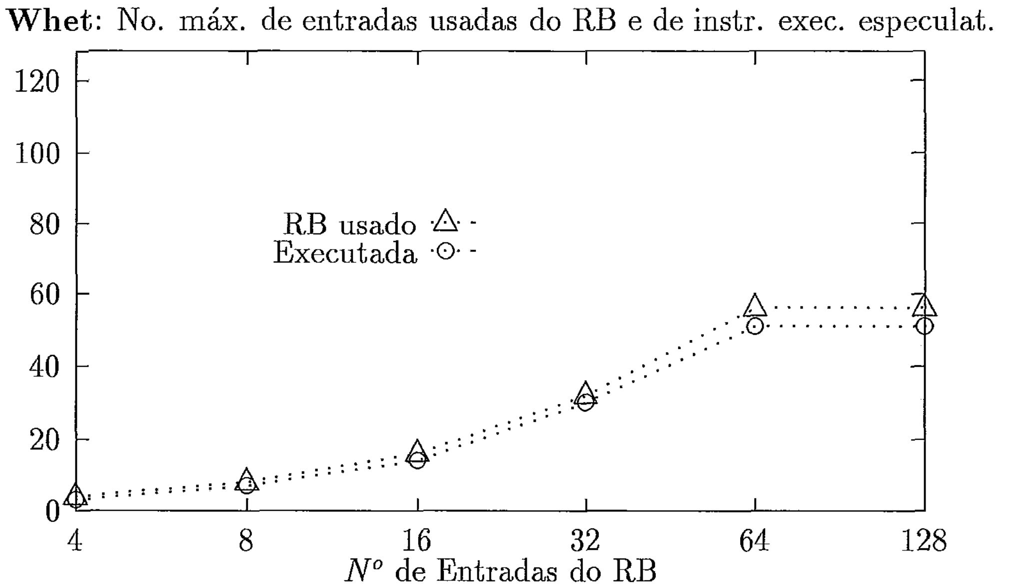


Figura 6.23: Número Máximo de Entradas do RB ocupadas e Número Máximo de Instruções Executadas Especulativamente (Whetstone).

do modelo regendo o critério "custo × desempenho", devemos obter medidas dos seus valores médios ao longo do tempo da execução. A seguir, apresentaremos as medidas dos valores médios realizadas que podem melhor ilustrar a capacidade de execução especulativa do modelo.

Inicialmente, analisamos as medidas realizadas no programa **Sed**, mostradas na Figura 6.24. As três curvas da Figura 6.24 representam, respectivamente, o número médio de entradas ocupadas do RB, o número de instruções despachadas e executadas especulativamente. Podemos notar que todas as curvas crescem à medida que o tamanho do RB aumentava, e estabilizam para RBs com mais de 64 entradas. Para o maior RB avaliado (128 entradas), o número médio de entradas do RB ocupadas foi de 20,34, bem inferior ao número máximo observado anteriormente. O número médio de instruções despachadas e executadas especulativamente foram 18,10 e 13,98, respectivamente.

O número médio de entradas do RB ocupadas é maior do que o

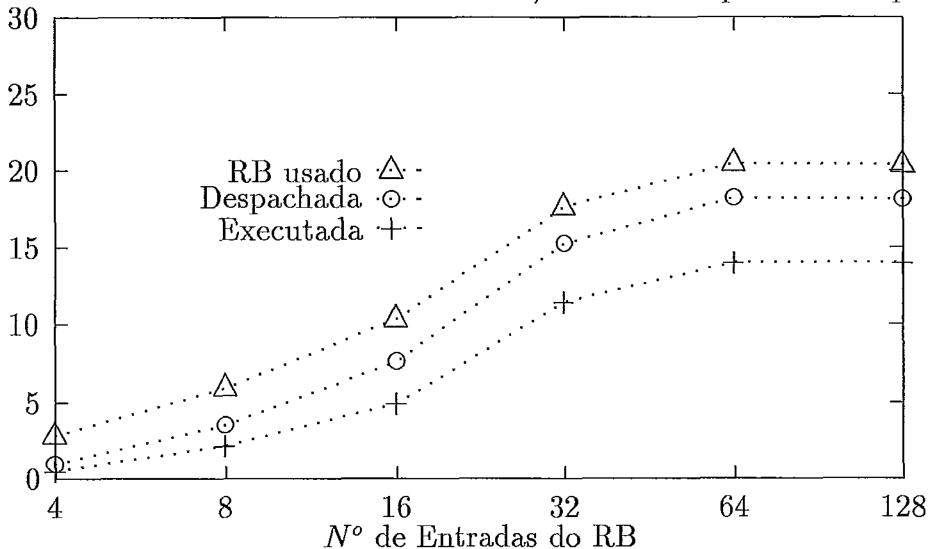


Figura 6.24: Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Sed).

número de instruções despachadas especulativamente, porque nem sempre as primeiras instruções na fila do RB são desvios condicionais não terminadas. As instruções antecedendo a primeira instrução de desvio condicional não terminada na fila não são consideradas especulativas. O número de instruções despachadas especulativamente, por sua vez, é maior do que o número de instruções executadas especulativamente. O motivo dessa diferença é que podem existir instruções despachadas especulativamente nas *reservation stations*, e que estão aguardando o início de suas execuções por causa de dependência de dados. Ressaltamos uma maior importância para o número de instruções executadas especulativamente, pois é ele que mede realmente o trabalho especulativo útil realizado pelo processador.

Para o programa **Espresso**, o nível da execução especulativa conseguido superou o do programa **Sed**. Pelo gráfico da Figura 6.25, em média, 19,69 instruções foram executadas especulativamente para um RB com 64 entradas, e 19.71 instruções para um RB com 128 entradas. Isso mostra que apenas um aumento insignificante foi obtido quando dobramos o tamanho do RB. Já para o programa

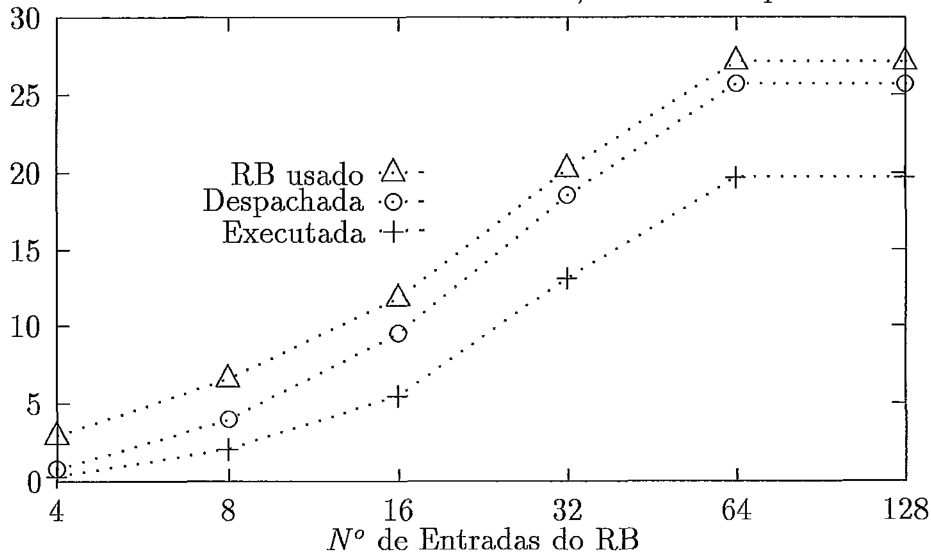


Figura 6.25: Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Espresso).

Whetstone, foi registrado o menor nível de execução especulativa dos programas de teste, alcançando, em média, apenas 4,72 instruções executadas especulativamente para RBs com 128 entradas (vide Figura 6.26). O motivo principal desse baixo desempenho foi a freqüente ocorrência de conflitos no uso de recursos, que limitou o número médio de instruções despachadas. Como a conseqüêcia dessa limitação, diminuiu também o número médio de instruções que o processador é capaz de executar especulativamente.

Apesar de cada programa de teste ter demonstrado um nível de execução especulativa diferente, nenhum deles apresentou um número médio de entradas do RB ocupadas maior do que 32, mesmo para RBs com 64 ou 128 entradas. Essa observação reforça a escolha anterior de um RB com 32 entradas como sendo a melhor alternativa para o nosso modelo, levando-se em conta o compromisso "custo × desempenho". Comparando os gráficos da taxa de aceleração e do número médio de instruções executadas especulativamente, podemos concluir que para atingir uma melhor taxa de aceleração, além de prover uma boa capacidade de execução espe-

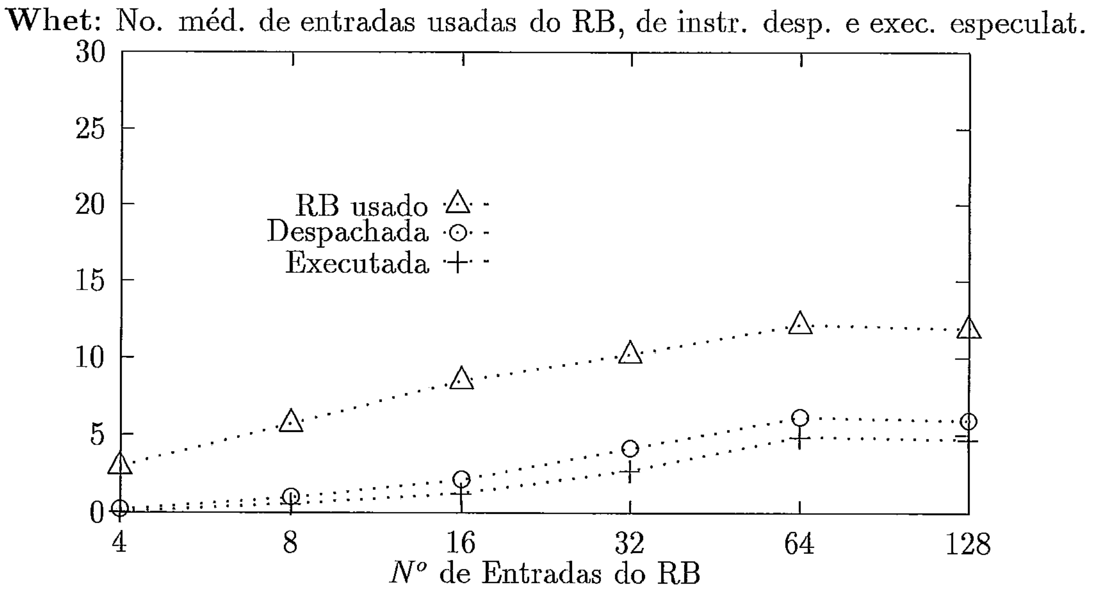


Figura 6.26: Número Médio de Entradas do RB ocupadas, de Instruções Despachadas e Executadas Especulativamente (Whetstone).

culativa, uma série de condições precisam ser atendidas: por exemplo, o programa em execução deve possuir um volume de paralelismo suficiente a ser explorado pelo processador. Por outro lado, o processador também deve dispor de recursos suficientes, de modo que as instruções despachadas especulativamente possam terminar mais rápido possível.

A seguir, faremos uma breve análise sobre o número de desvios condicionais que são despachadas especulativamente. A Figura 6.27 mostra o número máximo e o número médio de desvios condicionais no RB obtidos durante a execução do programa **Diff**. Esses números dão uma idéia aproximada do número de desvios condicionais embutidos que o processador consegue executar especulativamente. Para um RB com 32 entradas, em algum momento da execução, o processador chegou a alocar 7 desvios condicionais no RB, enquanto que, na média, foram alocados 2,93 desvios condicionais no RB. Para os programas que possuem uma grande percentagem de desvios entre suas instruções, tais como **Diff**, **Espresso**, **Grep** e **Sed**, o número médio de desvios condicionais alocados no RB se aproximou

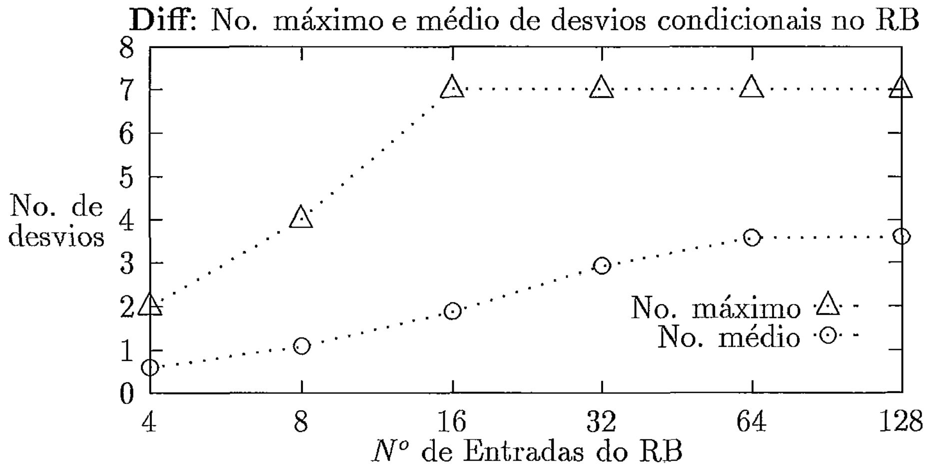


Figura 6.27: Número Máximo e Médio de Instruções de Desvio Condicional alocadas no RB (Diff).

de 3 num RB com 32 entradas. Para os programas **Compress**, **Li** e **Whetstone**, que possuem uma baixa percentagem de desvios nos seus *traces*, em média apenas pouco mais de 1 instrução de desvio condicional foi alocada no RB.

Capítulo 7

Conclusões

Neste trabalho, avaliamos o efeito da predição de desvios e do conceito de interrupções precisas no desempenho de um modelo de máquina Super Escalar realístico. O modelo é derivado do processador i860 da Intel, e incorpora um algoritmo de despacho associativo de múltiplas instruções. Mecanismos de previsão dinâmica de desvios, utilizando o *Branch Target Buffer* e o esquema de interrupções precisas com *Future File* (incluindo o *Reorder Buffer*) foram introduzidos no modelo Super Escalar.

Através da simulação do tipo *Trace-driven*, reproduzimos o comportamento desse modelo Super Escalar, variando diversos parâmetros dos mecanismos de predição de desvios e de interrupções precisas. Utilizando os resultados obtidos na simulação, identificamos importantes parâmetros da arquitetura que afetam o seu desempenho. Uma bateria de programas de teste provenientes do pacote SPEC e de utilitários do UNIX foi empregada durante nossos experimentos de avaliação do modelo Super Escalar.

Quanto ao mecanismo de predição de desvios condicionais, os principais parâmetros avaliados foram: o tamanho e a organização do *Branch Target Buffer*, o algoritmo de predição, o algoritmo de substituição e o algoritmo de alocação do *Branch Target Buffer*. Dentre as organizações, o BTB totalmente associativo apresentou o melhor desempenho. Variando o tamanho do BTB de 2 até 512 entradas, observamos que a taxa de acerto cresce à medida que aumentamos o tamanho do BTB. Taxas de acerto de até 99% foram obtidos para BTBs com mais

de 128 entradas para a maioria dos programas de teste. Foi constatado também que para BTBs com mais do que 256 entradas, todas as organizações (totalmente associativa, *2-way* e *4-way set-associative* e mapeamento direto) apresentaram taxas de *hit* semelhantes. Por outro lado, para BTBs com menos do que 32 entradas, a organização totalmente associativa produziu taxas de *hit* bastante superiores às taxas das demais organizações. Conforme indicado pelos experimentos, a taxa de aceleração depende de uma série de parâmetros da arquitetura, principalmente da taxa de acerto da previsão de desvios. A taxa de aceleração atingiu seu valor máximo quando a taxa de acerto da previsão de desvios se aproximou de 100%. A taxa de acerto, por sua vez, depende fortemente da taxa de *hit* do BTB e do algoritmo de predição de desvios utilizado pelo BTB. Quanto maior for a taxa de acerto, melhor será o desempenho da máquina.

Para os programas de teste utilizados, poucas diferenças entre as taxas de acerto dos algoritmos de previsão de 1 bit e 2 bits, foram registradas. Portanto, levando-se em conta o espaço de armazenamento necessário, o algoritmo de de 1 bit apresenta uma relação "custo × desempenho" mais vantajosa do que o de 2 bits. Tendo em vista que o algoritmo de substituição LRU explora melhor a localidade temporal dos desvios, ele conseguiu, na maioria das vezes, uma taxa de *hit* maior do que a do algoritmo aleatório. Por exemplo, para BTBs com 64 entradas, as taxas de *hit* obtidas pelo LRU é, em média, 4 vezes maior do que as do aleatório. Contudo, para BTBs com mais de 512 entradas (que conseguem armazenar a maioria das instruções de desvio encontradas no programa), a taxa de *hit* do BTB fica menos sensível ao tipo de algoritmo de substituição adotado (os dois algoritmos apresentaram praticamente as mesmas taxas de acerto).

A alta taxa de acertos nas previsões de desvios (acima de 98%) reduz sensivelmente o tempo de bloqueio do despacho de instruções que é causado pela penalidade de desvios (a percentagem do tempo da execução em que ocorreu bloqueio, nessa circunstância, cai para cerca de 1%), provocando um aumento na taxa de uso dos recursos da máquina. É importante que o processador disponha de recursos (*reservations stations* e unidades funcionais) suficientes para evitar que as oportunidades de execução em paralelo oferecidas pelo algoritmo de predição de

desvios sejam perdidas.

Tendo em vista o importante papel exercido pelo *Reorder Buffer* no conceito de interrupções precisas, decidimos investigar o efeito do seu tamanho no desempenho global, e na capacidade de execução especulativa do modelo. Ao monitorar a taxa de ocupação do RB observamos que, em média, não mais do que 32 entradas do RB foram requeridas durante a execução dos programas de teste. Para RBs com menos do que 32 entradas, ocorreram bloqueios freqüentes no despacho de instruções devido à falta de entradas livres no RB, impondo sérias limitações no desempenho. Embora mais instruções possam ser despachadas especulativamente se o RB contiver mais do que 32 entradas, ficou demonstrado que não se atinge taxa de aceleração significativamente maior. É preciso atingir um balanceamento entre a capacidade de armazenamento do RB e a quantidade de recursos do processador.

Para o tamanho da janela de instruções e a mistura de unidades funcionais (com suas *reservation stations* associadas) do nosso modelo (vide Capítulo 4), concluímos que um RB com 32 entradas é a melhor opção se levarmos em conta o compromisso "custo × desempenho". Para um RB com 32 entradas, verificamos que de 4,72 a 29,34 instruções foram executadas especulativamente em média, durante a execução dos programas de teste.

Na simulação e avaliação de um modelo de máquina, é fundamental que as medidas obtidas sejam validadas. Um dos métodos mais comuns de validação consiste em comparar o resultado da execução do programa de teste gerado pelo simulador com o gerado por uma outra máquina real. Devido à complexidade de modelagem e à limitação do tempo de processamento, o nosso simulador do modelo Super Escalar não executa efetivamente o programa. Na realidade, ele processa os eventos relevantes da execução do programa, contabilizando o tempo ocorrido em cada evento. Por esse motivo, a validação dos resultados da simulação torna-se mais difícil. Para resolver esse problema, utilizamos um programa extra de teste, cujo comportamento é bem conhecido, viabilizando desse modo a validação do nosso simulador (comparamos as medidas obtidas pelo simulador com as medidas previstas).

Para dar continuidade a este trabalho, podemos sugerir a implementação de um simulador de modelo Super Escalar que execute realmente o código do programa, gerando resultados da execução (ao invés de utilizar como entrada o *trace* do programa). Com o simulador real, o efeito das penalidades impostas pelas previsões erradas de desvios, poderia ser avaliado mais precisamente (pois nos *traces* de intruções, estão presentes apenas as instruções do caminho correto dos desvios). Uma outra sugestão seria a remodelagem da unidade funcional de *Load/Store*. No modelo utilizado, a unidade de *Load/Store* emprega um esquema de remoção de ambigüidades, que apesar de aumentar o *throughput* no acesso à memória, não garante o estado preciso na memória. Uma nova unidade de *Load/Store* que sempre mantivesse o estado em ordem na memória, utilizando informações do *Reorder Buffer*, poderia ser especificada. Adicionalmente, o efeito (no desempenho) da manutenção do estado em ordem na memória poderia ser investigado.

Referências Bibliográficas

[ACOST86] R. D. Acosta, J. Kjelstrup e H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," IEEE Transactions on Computers, Vol. C-35, No. 9, September 1986, pp. 815-828.

[ALPHA93] R. L. Sites, "Alpha AXP Architecture," Communications of the ACM, February 1993, Vol. 36, No. 2, pp. 33-44.

[ANDER67] D. W. Anderson, F. J. Sparacio e R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM Journal, No. 11, 1967, pp. 8-24.

[BARBO92] F. M. B. Barbosa e E. S. T. Fernandes, "Associative Dispatch of Multiple Instructions: A Study of its Effectiveness," Preliminar Version, Technical Report, Programa de Sistemas e Computação, ES-263/92, COPPE/UFRJ, June 1992, 18 pages.

[BUTLR91] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales e M. Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," Proceedings of the 18th Annual International Symposium on Computer Architecture, May 1991, pp. 276-286.

[CHAIT81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, e P. W. Markstein, "Register Allocation via Coloring," Computer Languages, Vol. 6, No. 1, 1981, pp. 47-57.

[CHAN91] P. P. Chang, W. Y. Chen, S. A. Mahlke e W. W. Hwu, "Comparing Static and Dynamic Code Scheduling for Multiple Instruction Issue Pro-

cessors," Proceedings of the 24th Annual International Symposium on Microarchitecture, MICRO-24, November 1991, pp. 25-33.

[CRAGO92] H. G. Cragon, "Branch Strategy Taxonomy and Performance Models," IEEE Computer Society Press, Los Alamitos, CA, USA, 1992

[DeROS87] J. A. DeRosa, e H. M. Levy, " An Evaluation of Branch Architectures," Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, pp.10-16.

[DIEFE92] K. Diefendorff, e M. Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," IEEE Micro, April 1992, pp. 40-63.

[DITZE87] D. R. Ditzel e H. R. Mclellan, "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," Proceedings of the 14th Annual International Symposium on Computer Architecture, 1987, pp. 2-9.

[DWYER87] H Dwyer e H. C. Torng, "A Fast Instruction Dispatch Unit for Multiple and Out-of-Sequence Issuances," Technical Report EE-CEG-87-15, School of Electrical Engineering, Cornell University, Ithaca, NY, USA, November 1987, 23 pages.

[FERNA93] E. S. T. Fernandes, M. B. Barbosa, e D. Simpson, "Evaluating the Cost of Conditional Branches on the Performance of Superscalar Machines,"

[FERNA92a] E. S. T. Fernandes e F. M. B. Barbosa, "Effects of Building Blocks on the Performance of Super-Scalar Architectures," Preliminar Version, Technical Report, Programa de Sistemas e Computação, ES-261/92, COPPE/UFRJ, January 1992, 18 pages.

[FERNA92b] E. S. T. Fernandes e F. M. B. Barbosa, "Effects of Building Blocks on the Performance of Super-Scalar Architectures," Proceedings of the 19th Annual International Symposium on Computer Architecture, ACM-SIGARCH and IEEE Computer Society, Australia, May 19-21, 1992, pp. 36-45.

[FERNA92c] F. M. B. Barbosa, " Efeito do Escalonamento Dinâmico no Desempenho de Processadores Super Escalares", Tese de Mestrado, Programa de Sistemas e Computação, COPPE/UFRJ, Fevereiro 1993, 76 páginas.

[FISHE81] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," IEEE Transactions on Computer, Vol. C-30, July 1981, pp. 478-490.

[FISHE84] J. A. Fisher, "VLIW Machine: A Multiprocessor for Compiling Scientific Code," IEEE Computer, July 1984, pp. 45-53.

[FLYNN67] M. J. Flynn, "The IBM System/360 Model 91: Some Remarks on System Development," IBM Journal, No. 11, 1967, pp. 2-7.

[FLYNN91] B. K. Bray, e M. J. Flynn, "Strategies for Branch Target Buffers," Proceedings of the 24th Annual International Symposium on Microarchitecture, MICRO-24, November 1991, pp. 42-50.

[GOODM87] A. R. Pleszkun, J. R. Goodman, W. C. Hsu, R. T. Joersz, G. Bier, P. Woest, e P. B. Schechter, "WISQ: A Restartable Architecture Using Queues," Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, pp.290-299.

[GROHO90] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor," IBM J. Res. Develop., Vol. 34, No. 1, January 1990, pp. 37-58.

[GROSS82] T. R. Gross, e J. L. Hennessy, "Optimized Delayed Branches," Proceedings of the 15th Annual Workshop on Microprogramming, MICRO-15, October 1982, pp. 114-120.

[HENNE86] J. L. Hennessy, "RISC-Based Processors: Concepts and Prospects," New Frontiers in Computer Architecture Conference Proceedings, March 1986, pp. 95-103.

[HOLGA80] R. W. Holgate e R. N. Ibbett, "An Analysis of Instruction Fetching Strategies in Pipelined Computers," IEEE Transactions on Computers, Vol. C-29, No. 4, April 1980, pp. 325-329.

[HWUPA87] W. W. Hwu, e Y. N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," Proceedings of the 14th Annual International Symposium on Computer Architecture, June 1987, pp. 18-26.

[INTEL89] Intel, "i860 64-Bit Microprocessor Programmer's Reference Manual," Intel, 1989.

[JAMES81] James E. Smith, "A Study of Branch Prediction Strategies," Proceedings of the 8th Annual International Symposium on Computer Architecture, May 1981, pp. 135-148.

[JAMES82] A. James, "Cache Memories," ACM Computing Survey, 14, No. 3, September 1982, pp. 473-530.

[JOHNS89] M. D. Smith, M. Johnson, e M. A. Horowitz, " Limits on Multiple Instruction Issue," Proceedings of the Third International Conference on Architectural Support for Programming Language and Operating Systems, April 1989, pp. 290-302.

[JOHNS91] M. Johnson, "Superscalar Processor Design," Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1991.

[JOUPP89] N. P. Jouppi, e D. W. Wall, "Available Instruction- Level Parallelism for Superscalar and Superpipelined Machines," Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989, pp. 272-282.

[KELLE75] R. M. Keller, "Look-Ahead Processors," Computing Surveys, Vol. 7, No. 4, December 1975, pp. 177-195.

[KOGGE81] Kogge, P. M. The Architecture of Pipelined Computers. New York: McGraw Hill, 1981.

[LAMMS92] M. S. Lam, e R. P. Wilson, "Limits of Control Flow on Parallelism," Proceedings of the 19th Annual International Symposium on Computer Architecture, June 1992, pp. 46-57.

[LEEJK84] J. K. F. Lee e A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," IEEE Computer, January 1984, pp. 6-21.

[LILJA88] D. J. Lilja, "Reducing the Branch Penalty in Pipelined Processors," IEEE Computers, July 1988, pp. 47-55.

[McFAR86] S. McFarling e J. Hennessy, "Reducing the Cost of Branches," Proceedings of the 13th Annual International Symposium on Computer Architecture, June 1986, pp. 396-403.

[McGEA90] S. McGeady, "Inside Intel's i960CA Superscalar Processor," Microprocessors and Microsystems, Vol. 14, No. 6, July/August 1990, pp. 385-396.

[PATTY85] Y. N. Patt, H. Hwu e M. C. Shebanow, "HPS, a New Microarchitecture: Rationale and Introduction," Proceedings of the 18th International Microprogramming Workshop, December 1985, pp. 103-108.

[PATTR81] D. A. Patterson, e C. H. Sequin, "RISC I: A Reduced Instruction Set Computer," Computer Architecture News, ACM-SIGARCH, Vol. 9, No. 3, May 1981, pp. 443-457.

[PLESZ85] J. S. Smith, e A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," Proceedings of the 12thz Annual International Symposium on Computer Architecture, June 1985, pp.36-44.

[PLESZ88] A. R. Pleszkun e G. S. Sohi, "The Performance Potential of Multiple Functional Unit Processors," Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp. 37-44.

[RAMAM77] C. V. Ramamoorthy e H. F. Li, "Pipeline Architecture," Computing Surveys, Vol. 9, No. 1, March 1977, pp. 61-102.

[SMITH82] A. J. Smith, "Cache Memories," ACM Computing Survey, Vol. 14, No. 3, September 1982, pp. 473-530.

[SPEC92] "SPEC Cint92 Release V.1.1 Technical Manual", System Performance Evaluation Cooperative, 1992.

[THORN64] J. E. Thornton, "Parallel Operation in the Control Data 6600," AFIPS Proceedings FJCC, pt 2, Vol. 26, 1964, pp. 33-40.

[TOMAS67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Journal, No. 11, 1967, pp. 25-33.

[TORNG84] H. C. Torng, "An Instruction Issuing Mechanism for Performance Enhancement," School of Electrical Engineering Technical Report EE-CEG-84-1, Cornell University, Ithaca, NY, February 1984.

[TORNG92] H. Dwyer, e H. C. Torng, "An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts," Proceedings of the 25th Annual International Symposium on Microarchitecture, MICRO-25, December 1992, pp. 272-281.

[VAJAP87] G. S. Sohi, e S. Vajapeyam, "Instruction Issue Logic for High-Performance Interruptable Pipelined Processors," Proceedings of the 14th International Symposium on Computer Architecture, June 1987, pp. 27-34.

[WARRE90] H. S. Warren Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," IBM J. Res. Develop., Vol. 34, No. 1, January 1990, pp. 85-92.

[WEISS84] S. Weiss e J. E. Smith, "Instruction Issue Logic for Pipelined Supercomputers," Proceedings of the 11th Annual International Symposium on Computer Architecture, June 1984, pp. 110-118.

[YEHTY91] T. Y. Yeh, e Y. N. Patt, "Two-Level Adaptive Training Branch Prediction," Proceedings of the 24th Annual International Symposium on Microarchitecture, MICRO-24, November 1991, pp. 51-61.

Apêndice A

Dados Obtidos na Simulação

A.1 Efeito do Tamanho e da Organização do BTB

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	2,43	2,43	2,44	2,58	2,86	3,14	3,27	3,43	3,45
compress	3,26	3,26	3,32	3,34	3,65	3,93	4,08	4,13	4,10
li	2,70	2,70	2,72	2,77	2,84	2,98	3,16	3,32	3,46
sed	2,25	2,25	2,32	2,74	2,76	2,79	2,90	3,05	3,17
grep	2,62	2,62	2,83	3,11	3,30	3,49	3,52	3,60	3,66
diff	2,74	2,74	2,77	3,13	3,35	3,56	3,68	3,75	3,78
whet	2,85	2,85	2,88	2,89	2,92	2,99	3,12	3,13	3,19

Tabela A.1: Taxa de Aceleração: BTB Mapeamente Direto, *All-Branch*, 2 bits, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	3,04	3,04	3,04	3,17	3,35	3,62	3,72	3,76	3,83
compress	3,30	3,30	3,30	3,45	3,51	3,87	4,05	4,10	4,10
li	2,73	2,73	2,73	2,80	2,89	3,07	3,26	3,40	3,55
sed	2,70	2,70	2,70	2,74	2,77	2,80	2,89	3,08	3,19
grep	2,83	2,83	2,83	3,17	3,47	3,49	3,52	3,60	3,74
diff	3,04	3,04	3,04	3,17	3,35	3,62	3,73	3,76	3,83
whet	2,85	2,85	2,85	2,88	2,91	3,00	3,11	3,18	3,18

Tabela A.2: Taxa de Aceleração: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	2,44	2,44	2,44	2,44	2,81	3,34	3,34	3,44	3,56
compress	3,30	3,45	3,45	3,45	3,62	3,81	4,03	4,10	4,10
li	2,73	2,77	2,77	2,77	2,89	3,03	3,30	3,44	3,57
sed	2,70	2,75	2,75	2,75	2,76	2,79	2,90	3,10	3,22
grep	2,83	2,90	2,90	2,90	3,47	3,49	3,52	3,62	3,74
diff	3,04	3,21	3,21	3,21	3,34	3,61	3,62	3,82	3,83
whet	2,85	2,90	2,90	2,90	2,95	3,01	3,16	3,21	3,22

Tabela A.3: Taxa de Aceleração: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	2,44	2,44	3,26	3,55	3,34	3,52	3,54	3,54	3,54
compress	3,30	3,45	3,46	3,98	4,10	4,10	4,10	4,10	4,10
li	2,73	2,77	2,89	3,04	3,30	3,50	3,64	3,66	3,66
sed	2,70	2,75	2,77	2,80	2,86	3,15	3,25	3,30	3,30
grep	2,83	2,90	3,47	3,50	3,52	3,62	3,76	3,79	3,79
diff	3,04	3,21	3,33	3,55	3,81	3,83	3,83	3,83	3,83
whet	2,85	2,90	2,93	3,00	3,14	3,22	3,22	3,22	3,22

Tabela A.4: Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	8,89	8,89	10,03	25,19	59,31	73,33	91,94	94,35	96,42
compress	11,95	11,95	18,99	20,28	46,46	63,58	79,14	88,86	98,89
li	12,01	12,07	15,49	21,15	28,60	45,30	60,42	79,76	88,89
sed	5,44	5,44	13,21	63,57	66,65	69.49	77,04	85,79	92,18
grep	12,39	12,39	40,66	60,52	74,74	87,53	89,59	93,04	95,71
diff	24,37	24,37	27,67	63,92	75,56	84,83	91,98	94,95	96,38
whet	14,43	14,43	17,14	19,33	22,03	47,95	69,64	79,79	94,20

Tabela A.5: Taxa de Acerto: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,96	9,96	9,96	15,86	64,45	70,31	92,01	94,92	97,98
compress	17,79	17,79	17,79	34,91	39,35	64,70	79,63	98,87	98,92
li	17,84	17,84	17,84	25,56	37,03	55,97	74,00	83,83	93,94
sed	61,27	61,27	61,27	63,19	66,82	69,47	76,91	86,72	94,91
grep	40,49	40,49	40,49	65,46	86,08	87,87	89,55	93,24	96,75
diff	58,36	58,36	58,36	66,03	76,03	89,79	96,15	97,61	99,41
whet	14,43	14,43	14,43	17,91	24,51	56,75	80,52	91,99	93,38

Tabela A.6: Taxa de Acerto: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,96	10,16	10,16	10,16	54,48	90,03	90,48	95,11	99,62
compress	17,79	35,10	35,10	35,10	47,93	61,32	92,14	98,88	98,83
li	17,84	23,87	23,87	23,87	39,22	54,02	78,67	88,02	95,17
sed	61,27	65,57	65,57	65,57	66,38	68,53	76,78	88,19	96,18
grep	40,49	50,18	50,18	50,18	86,29	88,04	89,47	94,26	96,98
diff	58,36	67,68	67,68	67,68	74,45	90,59	95,23	99,66	99,66
whet	14,43	19,85	19,85	19,85	42,50	56,82	86,47	96,13	98,93

Tabela A.7: Taxa de Acerto: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,96	10,16	83,78	87,42	90,54	98,95	99,80	99,81	99,81
compress	17,79	35,10	36,40	86,91	98,80	98,91	98,94	98,94	98,94
li	17,84	23,87	37,95	60,72	78,40	90,61	98,21	99,35	99,39
sed	61,27	65,57	66,49	68,59	72,83	92,86	97,63	99,62	99,63
grep	40,49	50,18	86,74	88,09	89,56	93,77	98,56	99,42	99,47
diff	58,36	67,68	75,53	87,42	99,26	99,73	99,78	99,78	99,78
whet	14,43	19,85	38,43	52,32	83,91	98,93	98,93	98,93	98,93

Tabela A.8: Taxa de Acerto: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,15	1,15	1,47	3,10	4,14	12,05	27,19	28,25	29,47
compress	9,61	9,61	9,80	9,95	18,23	21,05	24,19	26,16	29,97
li	6,68	6,68	6,74	7,05	7,42	8,63	11,40	15,81	17,83
sed	3,27	3,27	4,20	16,43	17,17	17,63	18,60	21,03	22,72
grep	0,91	0,91	6,20	9,77	16,73	22,58	23,17	24,18	25,12
diff	11,68	11,68	11,90	17,71	20,15	23,99	28,03	29,68	30,24
whet	32,10	32,10	32,51	32,64	32,88	35,86	39,19	41,32	43,51

Tabela A.9: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,40	1,40	1,40	1,55	8,51	10,21	27,31	28,47	30,32
compress	11,64	11,64	11,64	16,45	17,25	20,40	24,61	29,96	29,98
li	7,33	7,33	7,33	7,89	8,91	11,54	14,91	17,22	20,12
sed	15,97	15,97	15,97	16,31	17,39	17,66	18.61	21,36	24,10
grep	6,45	6,45	6,45	11,99	22,10	22,73	23,24	24,34	25,21
diff	17,19	17,19	17,19	18,72	20,50	26,97	29,81	30.60	32,58
whet	32,10	32,10	32,10	32,59	32,21	37,72	41,10	43,24	43,44

Tabela A.10: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,40	1,51	1,51	1,51	8,83	26,08	26,50	28,65	31,38
compress	11,63	16,45	16,45	16,45	17,85	19,55	28,87	29,96	29,98
li	7,33	8,00	8,00	8,00	9,19	11,23	17,23	19,09	20,28
sed	15,97	17,17	17,17	17,17	17,28	17,80	18,92	22,00	24,51
grep	6,45	8,43	8,43	8,43	22,10	22,83	23,22	24,57	25,35
diff	17,19	19,82	19,82	19,82	20,82	27,62	29,42	32,98	33,00
whet	32,10	32,75	32,75	32,75	36,75	37,98	42,35	43,98	44,41

Tabela A.11: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,40	1,51	23,16	26,99	26,57	30,95	31,48	31,49	31,49
compress	11,63	16,45	16,78	27,95	29,95	29,97	29,98	29,98	29,98
li	7,33	8,00	9,51	13,14	17,59	19,78	21,05	21,21	21,22
sed	15,97	17,17	17,41	17,93	18,80	23,78	25,05	25,56	25,56
grep	6,45	8,43	22,39	22,90	23,30	24,22	25,46	25,56	25,57
diff	17,19	19,82	22,36	26,99	32,71	32,98	33,01	33,01	33,01
whet	32,10	32,75	36,09	37,93	42,16	44,41	44,41	44,41	44,41

Tabela A.12: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	54,18	54,18	53,48	49,73	44,50	28,93	8,26	5,85	4,20
compress	45,79	45,79	43,21	42,69	27,88	19,50	12,13	6,25	0,78
li	38,42	38,42	37,80	36,44	34,39	29,49	21,96	13,62	8,37
sed	53,90	53,90	50,72	24,07	22,82	21,56	18,27	12,10	7,06
grep	57,79	57,79	46,95	36,69	24,56	10,65	9,39	6,81	4,46
diff	44,58	44,58	43,83	31,91	24,97	17,42	10,08	6,96	5,52
whet	20,28	20,28	19,47	19,31	18,69	13,64	8,66	5,26	1,69

Tabela A.13: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	53,56	53,56	53,56	52,52	37,51	34,16	8,00	5,56	2,56
compress	41,96	41,96	41,96	33,29	31,34	20,43	11,63	0,80	0,77
li	36,87	36,87	36,87	34,50	31,29	24,08	16,16	10,27	3,80
sed	25,58	25,58	25,58	24,09	22,17	21,37	18,50	10,92	4,46
grep	46,33	46,33	46,33	32,40	11,65	10,33	9,17	6,56	3,47
diff	33,33	33,33	33,33	29,70	24,85	13,31	5,24	3,61	0,70
whet	20,28	20,28	20,28	19,39	18,53	10,23	5,39	2,11	1,87

Tabela A.14: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	53,56	53,39	53,39	53,39	39,89	9,84	9,27	5,17	0,44
compress	41,96	33,17	33,17	33,17	27,04	22,55	4,46	0,80	0,80
li	36,87	34,96	34,96	34,96	30,54	25,30	12,04	7,05	3,20
sed	25,58	22,83	22,83	22,83	22,36	21,17	17,81	9,80	3,21
grep	46,33	42,14	42,14	42,14	11,51	10,15	9,16	5,55	3,19
diff	33,33	27,56	27,56	27,56	24,63	10,51	6,10	0,36	0,24
whet	20,28	19,07	19,07	19,07	11,82	9,49	3,17	0,81	0,10

Tabela A.15: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	53,56	53,39	14,67	14,14	9,20	1,08	0,20	0,19	0,19
compress	41,96	33,17	32,57	7,28	0,83	0,77	0,75	0,75	0,75
li	36,87	34,96	30,22	21,51	11,55	4,93	0,95	0,37	0,35
sed	25,58	22,83	22,06	20,76	18,30	5,51	1,85	0,29	0,28
grep	46,33	42.14	11,14	10,03	9,01	5,49	1,23	0,45	0,42
diff	33,33	25,56	21,90	12,14	0,80	0,29	0,22	0,22	0,22
whet	20,28	19,07	12,68	9,90	3,64	0,10	0,10	0,10	0,10

Tabela A.16: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Despacho por Desvios: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	8,11	8,11	9,58	24,74	57,86	73,01	91,63	94,32	94,42
compress	4,09	4,09	11,13	12,79	45,65	62,78	78,43	89,68	99,71
li	11,42	11,42	14,83	20,49	27,95	44,70	60,35	79,65	88,86
sed	5,15	5,15	12,91	63,29	66,38	69,22	76,77	85,51	91,92
grep	1,41	1,41	34,77	54,62	74,36	87,16	89,22	92,73	95,51
diff	24,35	24,35	27,65	63,90	75,55	84,82	91,98	94,95	96,38
whet	3,49	3,49	6,19	8,38	16,65	44,50	68,29	79,66	95,19

Tabela A.17: Taxa de *Hit* do BTB: Mapeamento Direto, *All-Branch*, 2 bits, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,51	9,51	9,51	15,41	64,11	69,97	91,97	94,90	97,98
compress	16,34	16,34	16,34	33,46	38,54	63,90	79,45	99,69	99,74
li	17,18	17,18	17,18	24,90	36,38	55,38	73,96	83,80	93,93
sed	60,98	60,98	60.98	62,90	66,55	69,20	76,64	86,44	94,88
grep	35,05	35,05	35,05	59,57	85,70	87,49	89,18	92,93	96,65
diff	58,34	58,34	58,34	66,01	76,03	89,78	96,14	97,61	99,41
whet	3,49	3,49	3,49	6,79	14,67	54,40	78,41	92,97	94.36

Tabela A.18: Taxa de *Hit* do BTB: *2-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,51	9,82	9,82	9,82	54,14	89,69	90,18	95,11	99,62
compress	16,34	33,67	33,67	33,67	46,50	60,52	93,95	99,69	99,75
li	17,18	23,21	23,12	23,21	38,62	53,41	78,63	87,99	95,16
sed	60,98	65,29	65,29	65,29	66,11	68,26	76,51	87,92	96,14
grep	35,05	44,31	44,31	44,31	85,91	87,65	89,10	94,01	96,78
diff	58,34	67,67	67,67	67,67	74,45	90,59	95,23	99,66	99,76
whet	3,49	8,90	8,90	8,90	34,14	54,04	86,46	97.11	99,91

Tabela A.19: Taxa de *Hit* do BTB: *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,51	9,82	83,44	87,42	90,26	98,93	99,30	99,81	99,81
compress	16,34	33,67	34,97	86,55	99,62	99.73	99,77	99,77	99,77
li	17,18	23,21	37,35	60,66	78,35	90,58	98,25	99,40	99,45
sed	60,98	65,29	66,22	68,32	72,56	92,64	97,62	99,62	99,63
grep	35,05	44,31	86,38	87,71	89,19	93,62	98,56	99,43	99,47
diff	58,34	67,67	75,52	87,42	99,26	99,72	99,78	99,78	99,78
whet	3,49	8,90	27,48	46,80	84,86	99,91	99,91	99,91	99,91

Tabela A.20: Taxa de *Hit* do BTB: Totalmente Associativo, *All-Branch*, 2 bits, LRU, *Miss-Taken*.

A.2 Impacto do Algoritmo de Previsão

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	2,44	2,44	3,26	3,35	3,35	3,52	3,54	3,54	3,54
compress	3,29	3,44	3,45	3,96	4,09	4,09	4,09	4,09	4,09
li	2,73	2,77	2,89	3,08	3,30	3,50	3,64	3,66	3,66
sed	2,70	2,75	2,77	2,80	2,86	3,15	3,25	3,30	3,30
grep	2,84	2,90	3,53	3,55	3,58	3,66	3,76	3,79	3,79
diff	3,04	3,21	3,44	3,51	3,81	3,83	3,83	3,83	3,83
whet	2,85	2,90	2,93	3,00	3,14	3,22	3,22	3,22	3,22

Tabela A.21: Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	10,72	10,92	84,54	90,96	91,26	99,00	99,80	99,81	99,81
compress	18,41	35,74	37,07	87,19	98,53	98,64	98.68	98,68	98,68
li	17,84	23,87	37,95	60,74	78,43	90,62	98,21	99,33	99,37
sed	61,27	65,57	66,49	68,59	72,82	92,70	97,65	99,62	99,63
grep	41,43	50,37	87,04	88,39	89,86	94,04	98,58	99,43	99,48
diff	58,36	67,68	75,53	87,43	99,26	99,73	99,78	99,79	99,79
whet	14,43	19,85	38,43	52,88	83,74	98,76	98,76	98,76	98,76

Tabela A.22: Taxa de Acerto: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,69	1,77	23,58	26,78	27,00	30,99	31,49	31,49	31,49
compress	11,80	16,60	16,93	27,67	29,72	29,74	29,74	29,75	29,75
li	7,33	8,00	9,51	13,14	17,58	19,78	21,04	21,20	21,20
sed	15,97	17,17	17,41	17,93	18,82	23,78	25,06	25,56	25,56
grep	6,56	8,55	22,13	22,65	23,07	24,12	25,47	25,57	25,58
diff	17,19	19,82	22,36	27,00	32,71	32,98	33,01	33,01	33,01
whet	32,10	32,75	36,09	37,97	42,12	44,37	44,37	44,37	44,37

Tabela A.23: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	52,96	52,79	13,74	8,63	8,30	1,01	0,19	0,18	0,18
compress	41,49	32,72	32,12	7,07	1,00	0,95	0,93	0,93	0,93
li	36,87	34,96	30,22	21,49	11,54	4,92	0,96	0,39	0,37
sed	25,58	22,83	22,06	20,75	18,30	5,49	1,82	0,29	0,28
grep	46,16	41,96	11,00	9,87	8,84	5,24	1,19	0,45	0,41
diff	33,33	27,56	21,90	12,14	0,80	0.28	0,22	0,22	0,22
whet	20,28	19,07	12,68	9,85	3,65	0,12	0,12	0,12	0,12

Tabela A.24: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	9,51	9,82	83,44	89,97	90,26	98,93	99,80	99,81	99,81
compress	16,34	33,67	34,97	86,55	99,62	99,73	99,77	99,77	99,77
li	17,18	23,21	37,35	60,66	78,35	90,58	98,25	99,40	99,45
sed	60,98	65,29	66,22	68,32	72,56	92,64	97,62	97,62	97,63
grep	35,05	44,31	86,36	87,71	89,19	93,62	98,56	99,43	99,47
diff	58,34	67,67	75,52	87,42	99,26	99,72	99,78	99,78	99,78
whet	3,49	8,90	27,48	46,80	84,86	99,91	99,91	99,91	91,91

Tabela A.25: Taxa de *Hit* do BTB: Totalmente Associativo, *All-Branch*, 1 bit, LRU, *Miss-Taken*.

A.3 Impacto do Algoritmo de Substituição

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	2,42	2,42	2,42	2,42	2,42	2,45	2,46	2,54	2,54
compress	3,26	3,26	3,26	3,26	3,26	3,26	3,26	4,10	4,10
li	2,70	2,70	2,70	2,70	2,70	2,70	2,91	3,28	3,66
sed	2,25	2,25	2,25	2,25	2,25	2,25	2,33	2,39	3,30
grep	2,62	2,63	2,62	2,62	2,62	2,62	2,67	2,69	2,85
diff	2,74	2,74	2,74	2,74	2,74	2,74	2,93	3,39	3,83
whet	2,85	2,85	2,94	3,00	3,11	3,22	3,22	3,22	3,22

Tabela A.26: Taxa de Aceleração: Totalmente Associativo, *All-Branch*, 2 bits, *Random*, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	8,89	8,89	8,90	8,90	8,91	10,62	12,10	99,80	99,81
compress	11,95	11,98	12,01	12,03	12,09	12,12	12,15	98,94	98,94
li	12,07	12,07	12,08	12,24	12,27	12,74	31,45	73,38	99,34
sed	5,44	5,44	5,45	5,47	5,47	5,49	14,58	19,16	99,63
grep	12,40	12,41	12,41	12,42	12,48	12,51	17,22	19,27	38,07
diff	24,73	24,73	24,38	24,39	24,39	24,85	46,82	79,22	99,78
whet	14,43	19,85	32,30	51,60	79,85	98,93	98,93	98,93	98,93

Tabela A.27: Taxa de Acerto: Totalmente Associativo, *All-Branch*, 2 bits, *Random*, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	1,15	1,15	1,15	1,16	1,16	1,46	1,59	31,49	31,49
compress	9,61	9,61	9,61	9,61	9,62	9,62	9,62	29,98	29,98
li	6,68	6,68	6,68	6,68	6,68	6,72	9,70	15,62	21,21
sed	3,27	3,27	3,28	3,28	3,28	3,28	4,51	5,06	25,56
grep	0,91	0,91	0,91	0,92	0,93	0,93	1,50	1,78	5,67
diff	11,68	11,68	11,68	11,68	11,68	11,79	15,14	24,39	33,01
whet	32,10	32,23	34,44	36,68	41,45	44,41	44,41	44,41	44,41

Tabela A.28: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	54,18	54,18	54,18	54,17	54,17	53,41	52,95	0,20	0,19
compress	45,79	45,79	45,78	45,78	45,77	45,76	45,75	0,75	0,75
li	38,41	38,41	38,42	38,39	38,38	38,25	30,81	14,15	0,37
sed	53,90	53,90	53,90	53,89	53,89	53,88	50,42	48,48	0,28
grep	57,79	57,79	57,79	57,79	57,76	57,75	56,20	55,42	47,00
diff	44,58	44,58	44,58	44,57	44,57	44,34	37,37	18,67	0,22
whet	20,28	20,10	16,96	13,60	4,99	0,10	0,10	0,10	0,10

Tabela A.29: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	8,11	8,11	8,12	8,13	8,17	9,85	11,33	99,81	99,81
compress	4,09	4,12	4,15	4,16	4,23	4,26	4,29	99,77	99,77
li	11,42	11,42	11,42	11,58	11,61	12,38	30,87	72,82	99,79
sed	5,15	5,15	5,17	5,19	5,22	5,22	14,31	18,69	99,63
grep	1,42	1,42	1,43	1,44	1,52	1,56	6,27	8,37	27,13
diff	24,35	24,35	24,36	24,38	24,39	24,84	24,82	79,22	99,78
whet	3,49	3,75	21,36	49,46	77,71	99,91	99,91	99,91	99,91

Tabela A.30: Taxa de *Hit*: Totalmente Associativo, *All-Branch*, 2 bit, *Random*, *Miss-Taken*.

A.4 Impacto do Algoritmo de Alocação

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	–	3,53	3,53	3,53	3,55	3,55	3,55	3,55	3,55
compress	–	4,04	4,04	4,04	4,04	4,04	4,04	4,04	4,04
li	–	3,67	3,67	3,67	3,67	3,67	3,67	3,67	3,67
sed	–	3,31	3,31	3,31	3,31	3,31	3,31	3,31	3,31
grep	–	3,73	3,73	3,73	3,80	3,80	3,80	3,80	3,80
diff	–	3,85	3,85	3,85	3,85	3,85	3,85	3,85	3,85
whet	–	3,22	3,22	3,22	3,22	3,22	3,22	3,22	3,22

Tabela A.31: Taxa de Aceleração: 4-way Set-Associative, Taken-Branch, 2 bits, LRU, *Miss-Not Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	–	98,28	98,28	98,28	98,98	98,98	98,98	98,98	98,98
compress	–	96,02	96,02	96,02	96,02	96,02	96,02	96,02	96,02
li	–	99,83	99,83	99,83	99,81	99,78	99,76	99,76	99,76
sed	–	99,99	99,99	99,99	99,99	99,99	99,99	99,99	99,99
grep	–	99,03	99,03	99,03	99,97	99,98	99,98	99,98	99,98
diff	–	99,99	99,99	99,99	99,99	99,99	99,99	99,99	99,99
whet	–	97,69	97,69	97,69	97,69	97,69	97,69	97,69	97,69

Tabela A.32: Taxa de Acerto: 4-way Set-Associative, Taken-Branch, 2 bits, LRU, *Miss-Not Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	–	30,21	30,21	30,21	30,21	31,55	31,55	31,55	31,55
compress	–	27,99	27,99	27,99	27,99	27,99	27,99	27,99	27,99
li	–	21,29	21,29	21,29	21,28	21,27	21,27	21,27	21,27
sed	–	25,64	25,64	25,64	25,64	25,64	25,64	25,64	25,64
grep	–	26,02	26,02	26,02	25,69	26,70	26,70	26,70	26,70
diff	–	33,06	33,06	33,06	33,06	33,06	33,06	33,06	33,06
whet	–	44,17	44,17	44,17	44,17	44,17	44,17	44,17	44,17

Tabela A.33: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	–	2,11	2,11	2,11	2,11	0,03	0,03	0,03	0,03
compress	–	2,63	2,63	2,63	2,63	2,63	2,63	2,63	2,63
li	–	0,12	0,12	0,12	0,13	0,15	0,16	0,16	0,16
sed	–	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
grep	–	1,43	1,43	1,43	0,04	0,03	0,03	0,03	0,03
diff	–	0,01	0,01	0,01	0,01	0,01	0,01	0,01	0,01
whet	–	0,19	0,19	0,19	0,19	0,19	0,19	0,19	0,19

Tabela A.34: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Desvios: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*.

N^o Ent. BTB	2	4	8	16	32	64	128	256	512
espresso	–	2,63	2,63	2,63	3,62	3,62	3,62	3,62	3,62
compress	–	15,92	15,92	15,92	15,92	15,92	15,92	15,92	15,92
li	–	0,77	0,77	0,77	0,86	0,90	0,92	0,92	0,92
sed	–	0,28	0,28	0,28	0,28	0,28	0,28	0,28	0,28
grep	–	10,32	10,32	10,32	10,97	10,98	10,98	10,98	10,98
diff	–	0,02	0,02	0,02	0,02	0,02	0,02	0,02	0,02
whet	–	13,21	13,21	13,21	13,22	13,22	13,22	13,22	13,22

Tabela A.35: Taxa de *Hit*: *4-way Set-Associative*, *Taken-Branch*, 2 bit, LRU, *Miss-Not Taken*.

A.5 Efeito do Tamanho do *Reorder Buffer*

N^o Ent. RB	4	8	16	32	64	128
espresso	2,95	3,20	3,65	3,72	3,51	3,51
compress	3,03	3,64	4,14	4,36	4,15	4,10
li	2,72	3,25	3,53	3,61	3,57	3,57
sed	2,70	3,12	3,44	3,36	3,22	3,22
grep	2,94	3,42	3,85	3,76	3,74	3,74
diff	2,93	3,64	3,73	4,05	3,83	3,82
whet	2,52	2,95	3,24	3,28	3,25	3,22

Tabela A.36: Taxa de Aceleração: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bits, LRU, *Miss-Taken.*

N^o Ent. RB	4	8	16	32	64	128
espresso	0,0	0,12	8,27	22,28	33,42	33,39
compress	0,0	6,50	10,75	22,27	27,58	29,93
li	0,0	2,20	10,61	17,50	20,41	20,23
sed	0,0	1,50	6,35	17,00	24,73	24,56
grep	0,0	0,32	7,13	17,81	25,30	25,40
diff	1,55	5,85	14,17	28,00	34,44	34,36
whet	2,84	21,83	37,71	42,92	44,13	44,41

Tabela A.37: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de Recursos: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken.*

N^o Ent. RB	4	8	16	32	64	128
espresso	35,06	44,73	31,25	13,14	0,03	0,02
compress	39,76	42,01	31,96	10,83	3,21	0,02
li	34,53	30,40	17,26	4,20	0,06	0,0
sed	35,74	38,68	28,08	10,36	0,09	0,05
grep	40,86	46,20	30,73	11,86	0,14	0,01
diff	45,91	48,00	32,72	8,55	0,26	0,19
whet	39,83	28,70	10,23	2,80	0,33	0,00

Tabela A.38: Percentagem do Tempo de Execução em que Ocorreu Bloqueio de Depacho por Falta de entradas no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken.*

N^o Ent. RB	4	8	16	32	64	128
espresso	4	8	16	32	64	84
compress	4	8	16	32	64	132
li	4	8	16	32	64	90
sed	4	8	16	32	64	79
grep	4	8	16	32	64	107
diff	4	8	16	32	64	128
whet	4	8	16	32	56	56

Tabela A.39: Número Máximo de entradas do RB ocupadas: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	2,92	6,58	11,83	20,24	27,10	27,11
compress	3,15	6,66	11,89	17,65	30,54	37,19
li	2,67	5,50	8,97	12,17	14,18	14,63
sed	2,83	5,96	10,35	17,62	20,44	20,34
grep	3,02	6,44	11,64	18,89	27,36	27,49
diff	3,06	6,44	11,62	18,86	25,68	26,84
whet	2,96	5,70	8,51	10,24	12,15	11,89

Tabela A.40: Número Médio de entradas do RB ocupadas: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	0,81	3,98	9,51	18,52	25,66	25,67
compress	0,45	2,25	6,75	13,03	27,89	34,93
li	0,47	1,79	4,94	8,54	10,67	11,22
sed	0,97	3,57	7,67	15,25	18,19	18,10
grep	0,89	3,45	8,98	16,34	25,34	25,47
diff	0,94	3,79	8,74	15,99	23,18	24,35
whet	0,21	1,01	2,14	4,12	6,14	5,92

Tabela A.41: Número Médio de Instruções Despachadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	3	7	15	31	62	81
compress	3	7	15	31	62	124
li	3	7	15	31	61	120
sed	3	7	15	31	61	86
grep	3	7	15	31	63	126
diff	3	7	15	31	63	127
whet	3	7	14	30	51	51

Tabela A.42: Número Máximo de Instruções Executadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	0,34	2,07	5,52	13,16	19,69	19,72
compress	0,25	1,42	4,04	8,81	22,46	29,34
li	0,25	0,95	2,75	5,44	7,32	7,87
sed	0,50	2,17	4,95	11,43	13,98	13,98
grep	0,49	2,02	5,09	11,35	19,76	19,94
diff	0,52	2,31	5,74	11,75	18,46	19,42
whet	0,10	0,60	1,27	2,69	4,89	4,72

Tabela A.43: Número Médio de Instruções Executadas Especulativamente: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	3	4	6	7	7	7
compress	2	3	5	7	7	7
li	2	4	7	7	7	8
sed	2	4	7	7	7	7
grep	2	4	6	7	7	8
diff	2	4	7	7	7	7
whet	2	3	4	4	7	7

Tabela A.44: Número Máximo de Desvios Condicionais no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.

N^o Ent. RB	4	8	16	32	64	128
espresso	0,61	1,50	2,29	3,35	3,80	3,80
compress	0,30	0,57	1,02	1,50	2,36	2,63
li	0,33	0,67	1,12	1,48	1,69	1,72
sed	0,58	1,15	1,77	2,87	3,49	3,50
grep	0,54	0,93	1,80	2,53	3,26	3,26
diff	0,60	1,09	1,89	2,93	3,54	3,60
whet						

Tabela A.45: Número Médio de Desvios Condicionais no RB: BTB com 512 entradas, *4-way Set-Associative*, *All-Branch*, 2 bit, LRU, *Miss-Taken*.