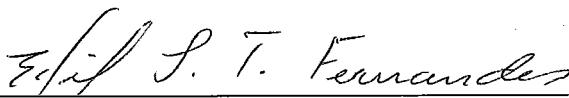


OTIMIZAÇÃO DO DESPACHO DE MÚLTIPLAS INSTRUÇÕES  
EM ARQUITETURAS SUPER ESCALARES

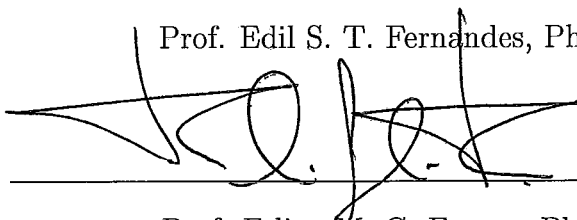
Gabriel Pereira da Silva

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

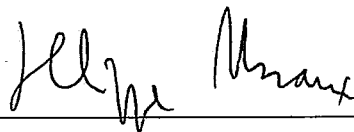
Aprovada por:



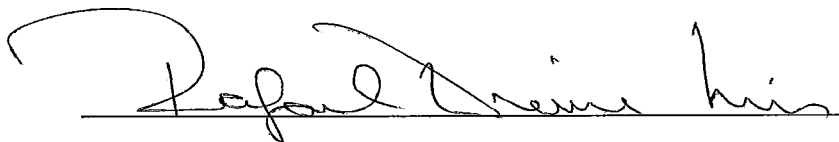
Prof. Edil S. T. Fernandes, Ph.D.



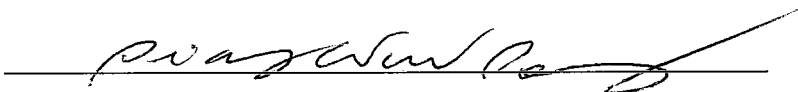
Prof. Felipe M. G. França, Ph.D.



Prof. Phillippe O. A. Navaux, Dr. Ing.



Prof. Rafael D. Lins, Ph.D.



Prof. Siang W. Song, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2000

SILVA, GABRIEL PEREIRA DA

Otimização do Despacho de Múltiplas Instruções em Arquiteturas Super Escalares [Rio de Janeiro] 2000

XIV, 192 pp., 29.7 cm, (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2000)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 – Arquiteturas Super Escalares

2 – Otimização do Despacho de Instruções

I. COPPE/UFRJ II. Título (série)

# DEDICATÓRIA

Ao nosso filho André Luiz,  
exemplo de coragem e determinação,  
com todo amor e carinho.  
(In Memoriam)

## AGRADECIMENTOS

Ao meu orientador, Prof. Edil S. T. Fernandes, pelo acompanhamento deste trabalho, com sua infatigável busca pela qualidade, mas também por toda sua amizade. Sem sua intervenção decisiva a realização deste trabalho não seria possível.

Ao Prof. Adilson Elias Xavier e aos membros da Comissão de Ensino da COPPE, pelo apoio e tolerância com os prazos, que permitiram a conclusão desta tese.

Aos funcionários e funcionárias da COPPE/Sistemas pela intervenção atenciosa e amiga nas horas de necessidade.

Ao NCE/UFRJ, pela apoio incondicional à realização desta tese.

À minha esposa Anna Paula, aos meus filhos Danilo e Caio Vítor, por todo amor e carinho que se constituíram apoio essencial durante os momentos difíceis.

A Deus e meus amigos do plano espiritual pela intervenção discreta, serena e precisa em todas horas.

A tantos outros que não pude mencionar, que colaboraram direta ou indiretamente, seja com atitudes, com uma palavra amiga, com um sorriso, com um pensamento ou uma prece, o meu muito obrigado.

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

## OTIMIZAÇÃO DO DESPACHO DE MÚLTIPLAS INSTRUÇÕES EM ARQUITETURAS SUPER ESCALARES

Gabriel Pereira da Silva

Setembro/2000

Orientador: Edil S. T. Fernandes

Programa: Engenharia de Sistemas e Computação

Esta tese investiga o “*despacho simultâneo de múltiplas instruções em arquiteturas super escalares.*” Considerando nosso interesse em tornar o mecanismo de despacho mais eficiente, dirigimos nossos esforços para o problema da detecção das dependências de dados.

Nossas atividades de pesquisa envolveram a concepção, implementação e avaliação da eficiência de mecanismos para otimizar o processo de detecção de dependências de dados entre um grande número de instruções que estão sendo despachadas em paralelo. Desenvolvemos três algoritmos de detecção de dependências de dados e um algoritmo de despacho, denominado “*Despacho Seletivo.*”

Os três primeiros algoritmos empregam diferentes tipos de memória *cache*, denominadas *caches* de dependências, que indicam como as instruções de um mesmo bloco básico interagem. Em tempo de execução, o algoritmo de despacho examina o conteúdo de sua *cache* de dependências, e na maioria das vezes, para cada instrução que está sendo despachada simultaneamente, o algoritmo determina imediatamente se existem ou não dependências de dados entre essa instrução e as demais.

O algoritmo de despacho seletivo, emprega quatro tipos de estações de reserva, e dependendo da instrução que está sendo examinada, ela será transferida para o tipo de estação apropriado. As estações de reserva são diferenciadas pela complexidade. Desse modo, especificamos estações que irão receber 3, 2, 1, ou zero resultados que ainda não foram produzidos. Ao especializarmos as estações de reserva, estaremos reduzindo a complexidade do *hardware* subjacente, e por esse motivo, teremos um processador mais rápido e eficiente.

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

## OPTIMIZING MULTIPLE INSTRUCTION DISPATCH IN SUPERSCALAR ARCHITECTURES

Gabriel Pereira da Silva

September/2000

Advisor: Edil S. T. Fernandes

Department: Computing and Systems Engineering

This thesis addresses the “*parallel dispatch of multiple instructions in superscalar architectures.*” We focus our efforts on *Data Dependence Detection* topic to improve the instruction dispatch mechanism of superscalar machines.

This work spanned through conception, implementation, and performance evaluation of hardware algorithms to detect data dependencies amongst several instructions which are dispatched concurrently. We developed three algorithms for the detection of data dependences and one Selective Dispatch algorithm.

The three detection algorithms employ special cache memories which store information regarding the interactions of instruction within the same basic block. At execution time, the dispatch algorithm examines the contents of its cache, and in the majority of times, interactions are determined immediately.

The selective dispatch algorithm uses four types of reservation stations, and depending on the instruction, it transfers the instruction to a suitable station. The reservation stations associated with the algorithm are of different levels of complexity. We specify stations that receive 3, 2, 1, and zero source operands. The provision of specialized reservation stations reduces the complexity of the underlying hardware, yielding a processor which is faster and more efficient.

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Trabalhos Anteriores</b>	<b>9</b>
2.1	Histórico . . . . .	9
2.2	O Algoritmo de Tomasulo . . . . .	14
2.2.1	Introdução . . . . .	14
2.2.2	As Estações de Reserva . . . . .	15
2.2.3	O Banco de Registradores . . . . .	16
2.2.4	A Difusão de Resultados no CDB . . . . .	17
2.2.5	Tratamento das Dependências de Dados . . . . .	18
2.2.6	Estudo de Caso . . . . .	20
2.3	Predição de Desvios . . . . .	21
2.3.1	Correlação Dinâmica de Desvio baseada em Caminhos . . . . .	21
2.3.2	Predição de Múltiplos Desvios . . . . .	24
2.4	<i>Trace Cache</i> . . . . .	28
2.5	Arquitetura Baseadas em Blocos . . . . .	33
2.5.1	Renomeação Estático/Dinâmica de Registradores . . . . .	33
2.5.2	Expansão de Bloco Atômico . . . . .	36
2.6	<i>Cache</i> de Renomeações . . . . .	40
2.7	Registradores Físico-Virtuais . . . . .	44
2.7.1	Introdução . . . . .	44
2.7.2	Renomeação de Registradores . . . . .	45
2.7.3	Descrição do Esquema . . . . .	46
2.7.4	Evitando o <i>Deadlock</i> . . . . .	49
2.7.5	Resultados . . . . .	50

2.8	Resumo . . . . .	50
<b>3</b>	<b>Os Modelos Propostos</b>	<b>51</b>
3.1	Introdução . . . . .	51
3.2	Eliminando As Dependências Falsas . . . . .	52
3.3	A Detecção das Dependências Verdadeiras . . . . .	54
3.4	Modelo Básico de Arquitetura . . . . .	57
3.5	Despacho Seletivo de Instruções . . . . .	60
3.6	<i>Cache</i> de Dependências . . . . .	63
3.6.1	Descrição do Buffer de Despacho . . . . .	64
3.6.2	<i>Cache</i> de Dependências Simples . . . . .	66
3.6.3	<i>Cache</i> de Dependências Inteligente . . . . .	68
3.6.4	<i>Cache</i> de Dependências Avançada . . . . .	69
3.7	Resumo . . . . .	71
<b>4</b>	<b>O Ambiente Experimental</b>	<b>73</b>
4.1	Introdução . . . . .	73
4.2	Vantagens e Desvantagens dos Métodos . . . . .	75
4.3	O Método Escolhido . . . . .	76
4.4	O Ambiente de Coleta de Dados . . . . .	79
4.5	Características dos Programas de Avaliação . . . . .	81
4.5.1	Introdução . . . . .	81
4.5.2	Programas de Ponto Flutuante . . . . .	82
4.5.3	Programas Inteiros . . . . .	86
4.6	Resumo . . . . .	90
<b>5</b>	<b>Resultados dos Experimentos</b>	<b>92</b>
5.1	Introdução . . . . .	92
5.2	Exploração de Paralelismo de Baixo Nível . . . . .	93
5.2.1	Introdução . . . . .	93
5.2.2	Distância Média entre Desvios Tomados . . . . .	93
5.2.3	Desvios e Blocos Básicos Estáticos de Maior Peso . . . . .	100
5.2.4	Localidade Espacial e Temporal . . . . .	102
5.2.5	Comentários . . . . .	103



5.3	Avaliação do Despacho Seletivo . . . . .	104
5.4	Avaliação da <i>Cache</i> de Dependências . . . . .	108
5.4.1	Largura de Despacho de 8 Instruções . . . . .	109
5.4.2	Largura de Despacho de 16 Instruções . . . . .	113
5.5	Resumo . . . . .	118
<b>6</b>	<b>Conclusões e Trabalhos Futuros</b>	<b>121</b>
6.1	Conclusões . . . . .	121
6.2	Trabalhos Futuros . . . . .	127
<b>7</b>	<b>Referências Bibliográficas</b>	<b>129</b>
<b>A</b>	<b>Técnicas para Avaliação do Desempenho de Arquiteturas Super Escalares</b>	<b>140</b>
A.1	Introdução . . . . .	144
A.2	Trabalhos Anteriores . . . . .	145
A.3	Metodologia . . . . .	147
A.4	A Escolha da Amostras . . . . .	148
A.5	Análise dos Resultados . . . . .	149
A.5.1	Espresso . . . . .	149
A.5.2	Whetstone . . . . .	151
A.5.3	Xlisp . . . . .	152
A.5.4	Tomcatv . . . . .	154
A.6	Conclusões . . . . .	156
A.7	Bibliografia . . . . .	158
<b>B</b>	<b>An Analytical Time and Area Model for On-chip Multiported Me- mories</b>	<b>159</b>
B.1	Introduction . . . . .	162
B.2	On-Chip Multiported Memories . . . . .	163
B.2.1	The Components . . . . .	163
B.2.2	Memory Design Parameters . . . . .	165
B.3	Modeling the Basic Parameters . . . . .	166
B.4	The Time Model . . . . .	168

B.4.1	The Decoder . . . . .	169
B.4.2	Wordline Driver . . . . .	169
B.4.3	Bitlines and Memory Cells . . . . .	170
B.4.4	Sense Amplifiers and Output Drivers . . . . .	174
B.4.5	Total Access and Cycle Time . . . . .	175
B.5	The Area Model . . . . .	176
B.6	Experimental Results . . . . .	179
B.6.1	Cycle Time Analysis . . . . .	179
B.6.2	Area Analysis . . . . .	181
B.7	Conclusions . . . . .	183
B.8	Bibliography . . . . .	183

## **C Proposta de Renomeação de Registradores em Arquiteturas Super**

<b>Escalares</b>	<b>184</b>	
C.1	Descrição da Estrutura . . . . .	184
C.2	Descrição do Funcionamento . . . . .	186
C.3	Evitando o <i>Deadlock</i> . . . . .	188
C.4	Avaliação . . . . .	189
C.5	Resumo . . . . .	191

# Lista de Figuras

1.1	Processador Super Escalar com Janela de Instruções Centralizada . . .	4
1.2	Processador Super Escalar com Janela de Instruções Distribuída . . . .	6
2.1	Unidades Funcionais e Estações de Reserva do IBM 360/91 . . . . .	15
2.2	Banco de Registradores do IBM 360/91 . . . . .	16
2.3	Barramento Global do IBM 360/61 . . . . .	17
2.4	Organização de Um Esquema de Correlação por Caminho . . . . .	23
2.5	Predição de Dois Desvios com apenas uma PHT . . . . .	25
2.6	Esquema de Funcionamento da <i>Trace Cache</i> . . . . .	29
2.7	Múltiplas Predições por Ciclo . . . . .	33
2.8	Instrução Única vs. Bloco Básico como Unidade Atômica . . . . .	34
2.9	Formato de Instrução para Suportar <i>Tags</i> Estáticos . . . . .	35
2.10	Combinando Blocos Atômicos em um Bloco Atômico Ampliado . . . .	38
2.11	Janela de Instruções Organizada em <i>Trace Lines</i> . . . . .	42
2.12	Renomeação de Registradores no Modo Tradicional . . . . .	43
2.13	Renomeação de Registradores com Uso de <i>Trace Lines</i> . . . . .	44
2.14	Tabelas de Mapeamento . . . . .	47
2.15	<i>Esquema do Reorder Buffer</i> . . . . .	48
3.1	Resolução de Anti-Dependência . . . . .	53
3.2	Resolução de Dependência de Saída . . . . .	54
3.3	Diagrama de Fluxo de Controle dos Blocos Básicos . . . . .	57
3.4	Modelo da Arquitetura Super Escalar Básica . . . . .	58
3.5	Armazenamento de Blocos Básicos no <i>Buffer</i> de Despacho . . . . .	59
3.6	Tempo de Acesso ao Banco de Registradores . . . . .	61
3.7	Verificação de Dependências no Despacho . . . . .	64

3.8	Modificação para uso da <i>Cache</i> de Dependências . . . . .	66
3.9	Controle das Dependências Internas . . . . .	66
3.10	Controle das Dependências Internas - Opção 1 . . . . .	69
3.11	Controle das Dependências Internas - Opção 2 . . . . .	70
4.1	Método de Simulação . . . . .	80
5.1	Distância Entre Desvios Tomados - gcc e li . . . . .	96
5.2	Distância Entre Desvios Tomados - compress e vortex . . . . .	97
5.3	Distância Entre Desvios Tomados - go e perl . . . . .	98
5.4	Distância Entre Desvios Tomados - m88ksim e jpeg . . . . .	99
5.5	Comparações por Instrução Despachada - <i>Buffer c/ 8</i> Instruções . . .	111
5.6	Comparações por Instrução Despachada - <i>Buffer c/ 8</i> Instruções . . .	114
5.7	Comparações por Instrução Despachada - <i>Buffer c/ 16</i> Instruções . .	116
5.8	Comparações por Instrução Despachada - <i>Buffer c/ 16</i> Instruções . .	119
A.1	Frequência das Instruções – Espresso . . . . .	150
A.2	Frequência das Instruções – Whetstone . . . . .	152
A.3	Frequência das Instruções – Xlisp . . . . .	154
A.4	Frequência das Instruções – Tomcatv . . . . .	156
B.1	Two Basic Memory Cells . . . . .	163
B.2	Components of a Multiported On-chip Memory . . . . .	168
B.3	Single-ended Sense Amplifiers . . . . .	174
B.4	Differential Sense Amplifier . . . . .	175
B.5	Time Delay for Two Memory Configurations . . . . .	180
B.6	Area Estimations for Two Memory Configurations . . . . .	182
C.1	Tabela de Mapeamento . . . . .	185
C.2	Esquema Simplificado do <i>Reorder Buffer</i> . . . . .	185
C.3	Organização da Estrutura de Renomeação . . . . .	186

# Lista de Tabelas

2.1	Despachando Instruções Concorrentemente . . . . .	36
2.2	Comparação de Desempenho . . . . .	44
3.1	Resumo das Dependências do Trecho do Programa <i>go</i> . . . . .	56
3.2	Total de Comparações para o Despacho . . . . .	57
4.1	Métodos para Coleta de <i>traces</i> . . . . .	74
4.2	Comparação dos Métodos para Coleta de <i>traces</i> . . . . .	77
4.3	Total de Instruções - Ponto Flutuante . . . . .	83
4.4	Número Médio de Instruções por Bloco Básico - Ponto Flutuante . . . . .	83
4.5	Total de Instruções - Prog. Inteiros . . . . .	87
4.6	Numero Médio de Instruções por Bloco Básico - Prog. Inteiros . . . . .	87
5.1	Distância Média Entre Desvios Tomados . . . . .	94
5.2	Desvios e Blocos Básicos Executados nos Programas . . . . .	101
5.3	Análise dos Desvios Condicionais (em milhares) . . . . .	102
5.4	Taxa de Acerto no Último <i>Buffer</i> de Despacho . . . . .	103
5.5	Total de Instruções Executadas - Prog. Inteiros . . . . .	104
5.6	Total de Instruções Executadas - Ponto Flutuante . . . . .	105
5.7	Total de Instruções por Número de Dependências - Prog. Inteiros . . . . .	105
5.8	Total de Instruções por Número de Dependências - Ponto Flutuante . . . . .	106
5.9	Percentual por Número de Dependências - Prog. Inteiros . . . . .	107
5.10	Percentual Instruções por Número de Dependências - Ponto Flutuante . . . . .	107
5.11	Comparações (em milhões) - Prog. Inteiros . . . . .	109
5.12	Comparações por Instrução Despachada - Prog. Inteiros . . . . .	110
5.13	Percentual de Comparações - Prog. Inteiros . . . . .	112
5.14	Comparações (em milhões) - Ponto Flutuante . . . . .	112

5.15	Comparações por Instrução Despachada - Ponto Flutuante . . . . .	113
5.16	Percentual de Comparações - Ponto Flutuante . . . . .	113
5.17	Comparações (em milhões) - Prog. Inteiros . . . . .	115
5.18	Comparações por Instrução Despachada - Prog. Inteiros . . . . .	115
5.19	Percentual de Comparações - Prog. Inteiros . . . . .	117
5.20	Comparações (em milhões) - Ponto Flutuante . . . . .	117
5.21	Comparações por Instrução Despachada - Ponto Flutuante . . . . .	118
5.22	Percentual de Comparações - Ponto Flutuante . . . . .	118
A.1	Programas de Teste . . . . .	148
A.2	Configurações de Arquitetura . . . . .	149
A.3	Estatísticas Espresso 1% . . . . .	149
A.4	Estatísticas Espresso 100% . . . . .	150
A.5	Ocupação das Unidades Funcionais (Espresso 1%) . . . . .	151
A.6	Ocupação das Unidades Funcionais (Espresso 100%) . . . . .	151
A.7	Estatísticas Whetstone 1% . . . . .	151
A.8	Estatísticas Whetstone 100% . . . . .	152
A.9	Ocupação das Unidades Funcionais (Whetstone 1%) . . . . .	153
A.10	Ocupação das Unidades Funcionais (Whetstone 100%) . . . . .	153
A.11	Estatísticas Xlisp 1% . . . . .	153
A.12	Estatísticas Xlisp 100% . . . . .	154
A.13	Ocupação das Unidades Funcionais (Xlisp 1%) . . . . .	155
A.14	Ocupação das Unidades Funcionais (Xlisp 100%) . . . . .	155
A.15	Estatísticas Tomcatv 1% . . . . .	155
A.16	Estatísticas Tomcatv 100% . . . . .	156
A.17	Ocupação das Unidades Funcionais (Tomcatv 1%) . . . . .	157
A.18	Ocupação das Unidades Funcionais (Tomcatv 100%) . . . . .	157

# Capítulo 1

## Introdução

Aumentar o desempenho dos processadores tem sido um dos objetivos mais perseguidos desde o advento da computação. Ao longo do tempo, de acordo com o estado da arte disponível, a arquitetura dos processadores sofreu diversas transformações.

O uso de arquiteturas super escalares, capazes de executar mais de uma instrução por ciclo, tem sido a opção mais utilizada no projeto dos novos processadores.

Este tipo de arquitetura detecta e extrai o paralelismo a nível de instrução automaticamente, sem que sejam necessárias modificações quer no código fonte, quer no código objeto dos programas. Estas características são fatores determinantes desta preferência.

Diversos projetos de pesquisa comprovam que existe um elevado potencial de paralelismo no código objeto de programas convencionais, que pode chegar a centenas de instruções executadas por ciclo [1, 2, 3, 4, 5, 6, 7, 8].

Contudo, vários obstáculos impedem os projetistas de maximizar o desempenho das arquiteturas super escalares. Entre esses, podemos destacar as dependências de controle e de dados [9], que limitam o grau de paralelismo a nível de instrução que pode ser extraído dos programas de aplicação.

As dependências de controle resultam do fluxo de controle do programa em execução. Por exemplo, quando da execução do comando:

```
if < cond > then  
    list1  
else  
    list2
```

Não podemos determinar precisamente qual dos ramos será executado, sem que saibamos a *priori* o resultado da avaliação de  $\langle \text{cond} \rangle$ .

As dependências de dados resultam do fluxo de dados durante a execução do programa. Por exemplo, considere o seguinte trecho de programa:

```
...      ...  
i.      A := B+C;  
j.      E := A+7;  
...      ...
```

Dois comandos de atribuição estão especificados no programa acima. O  $j$ -ésimo depende dos resultados produzidos pelo  $i$ -ésimo comando (a variável  $A$ , usada como fonte pela instrução  $j$ , é gerada pelo comando  $i$ ). Dada uma instrução  $i$ , se denominarmos  $In(i)$  o conjunto dos operandos fonte da instrução  $i$  e  $Out(i)$  o seu conjunto de operandos destino, podemos definir três tipos de dependências de dados:

1. A instrução  $j$  apresenta uma dependência de dados **verdadeira** (RAW – *Read After Write*) em relação à instrução  $i$ , quando

$$Out(i) \cap In(j) \neq \emptyset;$$

2. A instrução  $j$  é **anti-dependente** (WAR – *Write After Read*) da instrução  $i$ , quando

$$In(i) \cap Out(j) \neq \emptyset;$$

3. A instrução  $j$  apresenta uma dependência de dados de **saída** (WAW – *Write After Write*) em relação à instrução  $i$ , quando

$$Out(i) \cap Out(j) \neq \emptyset.$$

Esses três tipos de dependências impõem um seqüenciamento no início da execução das instruções envolvidas, i.e., a execução de uma instrução fica condicionada ao término de uma outra. Caso contrário, o resultado que seria usado como operando fonte por uma instrução, poderia ser precocemente destruído por uma outra, ou poderia não estar pronto ainda. Logo, isso impede que um maior número de instruções de um programa seja executado em paralelo.



Diversas investigações têm sido realizadas para superar essas limitações. No campo das dependências de controle, preditores cada vez mais sofisticados têm sido desenvolvidos [10, 11, 12, 13, 14, 15, 16], permitindo ao processador prever com maior precisão a direção do fluxo de execução dos programas.

Junto com esses preditores, o conceito de execução especulativa foi desenvolvido. Segundo esse conceito, ao invés de aguardar pelo término de uma instrução de desvio condicional, o processador escolhe um dos caminhos do desvio (conforme indicado pelo preditor de desvios) e inicia a execução especulativa daquelas instruções. Quando da conclusão do desvio, o mecanismo que implementa a especulação verifica se o caminho especulado era o correto. Se este for o caso, ganha-se alguns ciclos de máquina com a execução antecipada. Caso contrário, é necessário neutralizar os efeitos provocados pela execução indevida.

Além do banco normal de registradores (também chamado banco de registradores arquiteturais), a implementação do conceito de execução especulativa requer uma outra estrutura para armazenar temporariamente o valor dos registradores e de um *reorder buffer* que reorganiza a ordem de atualização registradores arquiteturais: uma entrada no *reorder buffer* é reservada toda vez que uma instrução é despachada; à medida que as instruções anteriores forem terminando, a instrução corrente vai avançando na direção do topo do *reorder buffer*; quando do término da instrução no topo, o banco de registradores arquiteturais é atualizado, como se as instruções estivessem sendo iniciadas, executadas e concluídas seqüencialmente.

As dependências falsas (i.e., dependências de saída e anti-dependências) são eliminadas através das técnicas de renomeação: ao detectar a presença de um desses dois tipos de dependências de dados, a técnica de renomeação usa um outro registrador para atuar como destino da instrução. Em seguida, a técnica corrige os operandos fonte das instruções subseqüentes de modo que o novo registrador seja usado.

No caso das dependências de dados verdadeiras, recentes trabalhos explorando a predição de valores e reutilização de código [17, 18, 19, 20], procuram reduzir a limitação imposta pela execução seqüencial.

Para garantir a busca eficiente de um grande número de instruções em paralelo, muitos esforços também têm sido realizados [21, 22, 23, 24, 25, 26, 27, 28, 29, 30].

Na realidade, este deve ser o primeiro obstáculo a ser superado para a obtenção de maiores níveis de paralelismo. Ou seja, sem que a largura de banda disponível para a busca de instruções seja aumentada convenientemente, não é possível executar um grande número de instruções em paralelo.

Todavia, as soluções para a busca de instruções e para o tratamento das dependências não são suficientes para garantir o bom desempenho de uma arquitetura super escalar que despache um grande número de instruções em paralelo. Para que haja um aumento final no desempenho, é necessário que todos estágios do *pipeline* de instruções do processador tenham uma melhoria correspondente.

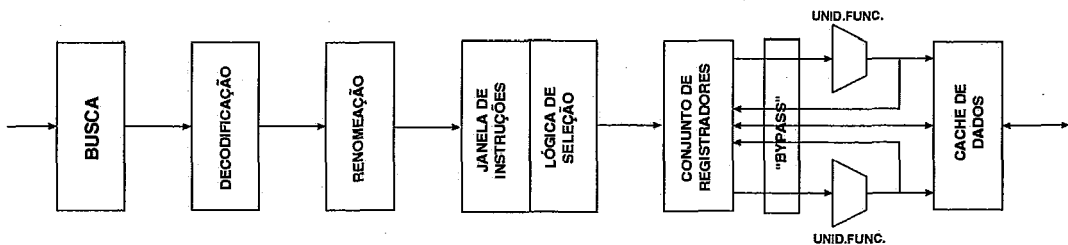


Figura 1.1: Processador Super Escalar com Janela de Instruções Centralizada

A Figura 1.1 apresenta o *pipeline* de instruções de um processador super escalar com janela de instruções centralizada. Estamos supondo que as instruções e os dados são armazenados em memórias *cache* separadas e que o processador utiliza tabelas de renomeação para eliminar as dependências falsas (i.e., dependências de saída e anti-dependências).

No início de cada ciclo, o estágio de busca da Figura 1.1 transfere algumas instruções da memória *cache* de instruções para o estágio de decodificação. Uma vez decodificadas, os seus operandos fonte e destino são renomeados pelo estágio subsequente. Em seguida, as instruções são transferidas para a janela de instruções do estágio de despacho.

Compete ao estágio de despacho a tarefa de escolha das instruções que podem ser executadas [31]. Desse modo, o conteúdo dos registradores que atuarão como operandos fonte são transferidos para as unidades funcionais do estágio de execução.

Os resultados produzidos pelas instruções são transferidos, pelo estágio de conclusão, para os registradores de renomeação. Finalmente, quando as instruções que produziram os resultados chegarem ao topo do *reorder buffer*, eles são escritos nos

registradores arquiteturais.

Os estágios de renomeação, despacho e conclusão são críticos quando se despacha um grande número de instruções em paralelo (e.g., mais que 4 instruções por ciclo). Alguns processadores, como o MIPS R10000 [32] e Alpha 21264 [33], que possuem uma largura de despacho de 4 instruções, utilizam o esquema de renomeação e despacho descrito acima. Esta organização, com janela de instruções centralizada e tabela de renomeação, possui as seguintes deficiências quando do despacho de um número maior de instruções:

- São necessários  $N \times (N - 1)$  comparadores para determinar as dependências verdadeiras entre as  $N$  instruções que estão sendo despachadas simultaneamente [34];
- A tabela de renomeação possui um número de portas de leitura que deve ser diretamente proporcional ao número de instruções despachadas em paralelo. Por outro lado, o banco de registradores precisa de  $2 \times N$  portas de leitura para fornecer os operandos requisitados pelas  $N$  instruções despachadas, além de  $N$  portas de escrita, uma para cada instrução concluída cujo resultado se destina ao banco de registradores. Conforme aumenta o número de portas, também aumenta o tempo de acesso a essas estruturas [35];
- O número de registradores físicos necessários para a renomeação aumenta consideravelmente conforme aumenta a largura de despacho. Este aumento, aliado ao grande número de portas, torna o tempo de acesso a esses dispositivos proibitivo para um bom desempenho da arquitetura [35, 36];
- O processo de identificação das instruções que estão prontas para serem executadas, requer um intervalo de tempo que é proporcional a  $N^2$  e a  $J^2$ , onde  $J$  é o total de instruções que estão na janela centralizada e  $N$  o número de instruções despachadas em paralelo [37];
- A operação de *bypass* dos resultados produzidos pelas unidades funcionais e que são transferidos para a entrada das unidades, como forma de diminuir em um ciclo o tempo de espera por esses operandos fonte, é uma operação cujo *hardware* envolvido é proporcional a  $F^2$ , onde  $F$  é o número de unidades funcionais existentes [37].

Na busca de uma possível alternativa para essas deficiências, destacamos o algoritmo de Tomasulo [38], que tem sido utilizado até hoje pelas arquiteturas de processadores. Esse algoritmo, desenvolvido em 1967, realiza o despacho de instruções para as unidades funcionais do IBM 360/91, mesmo que seus operandos fonte ainda não tenham sido gerados por instruções prévias.

Concebido originalmente para o despacho de uma única instrução por ciclo, o algoritmo de Tomasulo não pode ser usado no despacho múltiplas instruções por ciclo. Por esse motivo, alguns fabricantes modificaram o algoritmo de Tomasulo de modo que o despacho de múltiplas instruções por ciclo fosse viabilizado. Exemplos de arquiteturas super escalares que adotaram esta estratégia são o PowerPC 604 [39], Pentium Pro [40] e HAL SPARC64 [41].

Essas arquiteturas são caracterizadas pelo uso de uma janela de instruções distribuída. Elas empregam uma tabela de renomeação de registradores para eliminar as dependências falsas, embora isto não seja necessário no algoritmo original de Tomasulo. Para reduzir o efeito provocado pelas instruções de desvio condicional, elas também implementam o conceito de execução especulativa [42]. Um esquema de organização dessas arquiteturas pode ser visto na Figura 1.2.

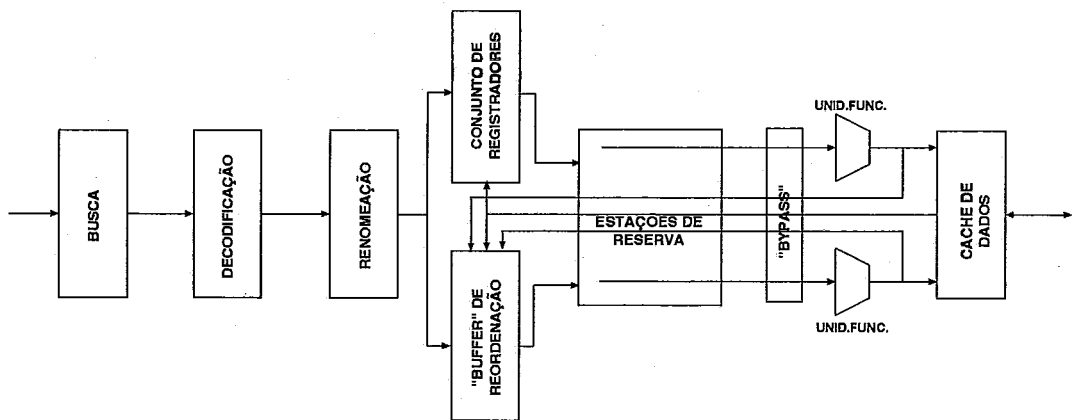


Figura 1.2: Processador Super Escalar com Janela de Instruções Distribuída

A principal diferença desse tipo de organização é que, no estágio de despacho, tanto as instruções como os seus operandos são copiados nas estações de reserva, formando a janela de instruções distribuída. Se algum dos operandos não estiver disponível, é enviado no seu lugar um *tag*, indicando qual a estação de reserva que irá produzir este operando. Quando os resultados forem produzidos pelas unidades

funcionais, eles são copiados também nas estações de reserva correspondentes.

Nas arquiteturas citadas [39, 40, 41], as entradas do *reorder buffer* incluem um campo para armazenar o resultado que a instrução irá produzir. Para localizar os operandos necessários às instruções que estão sendo despachadas e que ainda não foram transferidos para os registradores arquiteturais, é preciso realizar uma busca associativa no *reorder buffer* dessas arquiteturas.

Como ocorreu anteriormente com as máquinas com janela de instruções centralizadas, esse tipo de arquitetura também possui deficiências para o despacho amplo de instruções:

- Ainda são necessários  $N \times (N - 1)$  comparadores para determinar as dependências verdadeiras entre as  $N$  instruções que estão sendo despachadas simultaneamente;
- O número de portas de leitura da tabela de renomeação aumenta com o número de instruções despachadas a cada ciclo. O banco de registradores precisa de  $2 \times N$  portas de leitura para fornecer os operandos necessários às  $N$  instruções despachadas e  $N$  portas de escrita, uma para cada resultado produzido em cada ciclo;
- O algoritmo de Tomasulo requer barramentos de resultados e um comparador para cada *tag* armazenado nas estações de reserva, para que os resultados das instruções possam ser transmitidos das unidades funcionais para as estações de reserva. A medida que aumenta o número de instruções concluídas a cada ciclo, aumenta também o número de barramentos e comparadores necessários;
- Para aumentar o número de instruções que podem ser executadas em paralelo precisamos de um *reorder buffer* com um número maior de entradas (para armazenar temporariamente um número maior de instruções). Esse grande número de entradas torna o mecanismo de busca associativa mais complexo, com um impacto negativo no desempenho do processador super escalar.

Em vista disto, estamos apresentando nesta tese algumas modificações no algoritmo de Tomasulo [38], viabilizando o despacho amplo de instruções, com as seguintes características:

- O uso de um *future file* e de um *reorder buffer*, para implementar a execução especulativa. Uma das vantagens deste esquema é que ele dispensa a busca associativa, porque os resultados especulativos ficam armazenados no *future file*. Quando as instruções chegam ao topo do *reorder buffer*, seus resultados são transferidos para o banco de registradores arquiteturais;
- Três modelos de *cache* de dependências, que reduzem o número de comparações para a detecção de dependências verdadeiras quando do despacho de várias instruções por ciclo;
- Um mecanismo de despacho seletivo, que reduz o número de barramentos de resultado e de comparadores nas estações de reserva. Este mecanismo utiliza estações de reserva especializadas, com um número de comparadores proporcional ao número de operandos requeridos pelos diversos tipos de instrução.

A adoção destas modificações implica em um aumento do desempenho final da arquitetura, por causa da redução na complexidade do *hardware* diretamente envolvido nos estágios de renomeação, despacho e conclusão.

Particularmente, o uso de um menor número de portas no banco de registradores, estruturas de renomeação e *reorder buffer*, permite uma redução do seu tempo de acesso, o que resulta em ciclos de máquina mais rápidos.

Esta tese está organizada em seis capítulos: no Capítulo 2, os trabalhos anteriores, incluindo o algoritmo de Tomasulo, são apresentados em detalhes. No capítulo seguinte, apresentamos os modelos propostos originalmente nesta teste, ou seja, o uso de *future file* com *reorder buffer*, a *cache* de dependências e o mecanismo de despacho seletivo. O Capítulo 4 descreve o ambiente experimental junto com os programas de avaliação utilizados. No Capítulo 5 os resultados obtidos são apresentados e analisados. Finalmente, no Capítulo 6, são apresentadas as conclusões e trabalhos futuros.

# Capítulo 2

## Trabalhos Anteriores

### 2.1 Histórico

Nesta seção apresentamos um histórico dos trabalhos relacionados com o despacho amplo de instruções e, posteriormente, destacamos os trabalhos que consideramos importantes.

A maior parte desses trabalhos aborda os mecanismos de predição de múltiplos desvios por ciclo e mecanismos para aumentar a largura de banda disponível para busca de instruções. Em menor número, porém não menos importantes, estão os trabalhos de otimização dos estágios de renomeação, despacho e conclusão do *pipeline* de instruções.

Discutiremos inicialmente o algoritmo de Tomasulo [38], desenvolvido em 1967 para o sistema IBM 360/91. Ele trata do despacho seqüencial de instruções. Várias arquiteturas super escalares da atualidade adaptaram o algoritmo de Tomasulo para o despacho de múltiplas instruções por ciclo.

Na área de predição de desvios condicionais, destaca-se o trabalho de Yeh [43], que apresentou o conceito de preditores por correlação, que aumentou significativamente a acurácia dos esquemas de predição de desvio. A eficiência de preditores desse tipo se justifica porque o resultado de um desvio que está sendo executado normalmente está correlacionado com o resultado de outros desvios anteriores [44].

A maioria dos esquemas de predição de desvio utiliza uma tabela denominada PHT (*Pattern History Table*), composta tipicamente por contadores saturados de dois *bits* [45]. Os contadores recebem esta denominação porque eles não são modi-

ficados após alcançar o limite superior (valor 3) ou inferior (valor zero). A maneira mais simples de associar um contador com uma instrução de desvio é utilizar alguns *bits* do endereço da instrução de desvio, usualmente os *bits* menos significativos, para indexar a PHT.

O preditor por correlação [43, 46, 47, 48] utiliza um registrador de deslocamento, denominado BHR (*Branch History Register*). Ele é atualizado pela inclusão de um *bit* '1' se o desvio for tomado e um *bit* '0' em caso contrário. Este preditor pode ser global, e neste caso existe um único BHR, que é atualizado por todos os desvios e que pode ser combinado com alguns *bits* do endereço da instrução de desvio para formar um índice para a PHT.

Preditores que utilizam dois níveis de histórico (um nível global e outro local) podem endereçar a tabela PHT de duas maneiras principais: quando o BHR é utilizado sozinho para indexar a PHT, o preditor é chamado de GAg [43]; quando uma operação de “ou-exclusivo” é usada para combinar os *bits* do BHR com o endereço da instrução de desvio, o preditor é chamado de GSHARE [49]. O preditor GSHARE costuma ter melhor acurácia de predição.

Uma nova proposta de Yeh [11] estendeu o conceito de preditor por correlação para permitir a predição de múltiplos desvios por ciclo e a busca simultânea de vários blocos básicos.

Outros trabalhos nesta área que se destacam são a proposta de Franklin e Dutta [50] de um mecanismo de predição de desvio por sub-grafos orientados, que usa um histórico local para formar uma predição que codifica múltiplos desvios; o de Wallace e Nader [51, 52], que abrange mecanismos para a predição e busca de múltiplos blocos por ciclo; o de Sez nec et al. [53], com um método inovador para a busca e predição de até dois blocos básicos por ciclo. Rotenberg et al. [26] também adaptaram o GAg para investigar arquiteturas contendo a estrutura *trace cache*.

Mais recentemente, Patel et al. [54] apresentaram um preditor de múltiplos desvios para arquiteturas com *trace cache*. Esse preditor tenta conciliar as vantagens de um preditor GSHARE e também provê múltiplas predições por ciclo. O preditor realiza uma operação de “ou-exclusivo” do BHR com o endereço da primeira instrução do *trace* para indexar a PHT. Os autores modificaram as entradas da PHT que incluem agora múltiplos contadores de 2 *bits* para permitir a predição simultânea de



múltiplos desvios. Esse preditor tem acurácia superior quando comparado com o preditor multiportas GAg [11], mas não alcança a acurácia de um preditor simples GSHARE [49].

O artigo de Seznec et al. [53] apresenta um esquema de predição de dois blocos em avanço – TBA (*Two Block Ahead*). Em esquemas tradicionais, a informação relacionada com um bloco básico é usada para predizer o endereço do próximo bloco básico. No esquema proposto, a informação associada com um bloco básico é utilizada para predizer o endereço do bloco básico **após** o próximo bloco básico.

O TBA fornece a mesma acurácia dos esquemas convencionais de predição e com custos equivalentes de armazenamento. Como consequência, o TBA dobra a largura de banda para busca das instruções quando as estruturas de predição e a *cache* de instrução possuem duas portas de leitura. Entretanto, estruturas de armazenamento com duas portas de leitura ocupam mais área e tendem a ser mais lentas [35] e, por este motivo, o uso de estruturas com *interleaving* é recomendado. Os resultados obtidos são comparáveis aos obtidos com a *trace cache*, quando esta possui capacidade inferior a 128 Kbytes.

Nair [55] propôs uma predição baseada em “caminhos”, um esquema de predição de desvios por correlação que tem apenas um registrador de histórico (BHR) e uma tabela de histórico de padrões (PHT). A inovação é que a informação armazenada no BHR não é o resultado dos desvios anteriores, mas o valor truncado do endereço das instruções de desvio. Para fazer uma predição, alguns *bits* de cada endereço no registrador de histórico, assim como alguns *bits* do endereço atual, são concatenados para formar o índice da PHT. Então, um desvio é predito usando o conhecimento da seqüência, ou caminho, dos endereços das instruções de desvio que os levaram até ele. Isto dá ao preditor informações mais específicas sobre o fluxo de controle anterior do que um histórico do tipo “tomado/não tomado” dos resultados dos desvios. Jacobson et al. [56] refinaram o esquema de predição baseada em caminhos e o aplicaram em um preditor de próxima tarefa para processadores multi-escalares.

Jacobson et al. [57] também realizaram a adaptação deste esquema para uso em *trace caches*, substituindo o preditor convencional de desvios, o *buffer* de alvos de desvio (BTB) e a pilha de endereços de retorno (RAS). O preditor obtido tem acurácia melhor que preditores de desvio por correlação convencionais.

Mecanismos de predição de desvio acurados e *cache* de instruções com baixa taxa de falhas são essenciais para a operação eficiente das unidades de busca de arquiteturas super escalares com despacho amplo de instruções. Contudo, estas técnicas não são suficientes. A unidade de busca tem que extrair continuamente múltiplas instruções, não seqüenciais, da *cache* de instruções, realinhando-as na ordem adequada e encaminhando-as para decodificação.

O artigo de Conte [23], explora soluções para este problema e apresenta diversos esquemas com graus variáveis de custo e desempenho. O esquema mais geral, chamado de *collapsing buffer*, alcança desempenho quase perfeito e é capaz de alinhar as instruções consistentemente, em mais de 90% do tempo, sobre uma grande variação de larguras de despacho.

Nos métodos propostos por Yeh et al. [11] e Conte et al. [23], depois da busca de instruções da *cache*, havia ainda a necessidade de selecionar, alinhar e combinar blocos de instruções de diferentes linhas da *cache*. Esta lógica é complexa e introduz atrasos no *pipeline* principal, os quais a *trace cache* [26, 58] pretende remover. A *trace cache* combina blocos de instruções (*traces*) antes de armazená-los na *cache*. Na próxima vez que o endereço inicial de um *trace* for referenciado, eles podem ser lidos como um bloco e alimentados no *pipeline* sem ter que passar por nenhuma lógica complexa.

O trabalho de Stark et al. [59] apresenta um mecanismo original para evitar paradas do processador no caso de uma falha na *cache* de instruções: as instruções são despachadas fora-de-ordem para as estações de reserva. Neste caso, a busca de outras instruções na *cache* de instruções, decodificação e distribuição prossegue enquanto a falha na *cache* é atendida. Para isto, propõe o uso de uma *mask cache*, com maior capacidade que a *cache* de instruções, que guarda o registrador que é alterado por cada instrução. Este procedimento determinaria que instruções dependem de resultados produzidos por instruções que estão sendo buscadas para servir a falha da *cache*. Apenas instruções que não dependessem destes resultados seriam distribuídas para execução.

A necessidade de enfrentar o problema da renomeação dos registradores levou Sprangle e Patt [60], em 1994, a propor uma nova arquitetura em que a unidade de trabalho é o bloco básico, com extensões para permitir, além da renomeação

dinâmica, a renomeação estática de registradores. A vantagem desta nova organização é que ela reduz o número de comparadores requeridos pelo estágio de despacho, assim como o número de portas do banco de registradores. O modelo apresentado nesta tese, ao contrário do trabalho de Patt, não requer a modificação do código objeto e é realizado dinamicamente.

Posteriormente, Melvin e Patt [61] apresentaram a proposta de uma arquitetura estruturada em blocos *atômicos*. Essa arquitetura trata um grupo inteiro de instruções com uma unidade atômica, assim como as arquiteturas convencionais tratam instruções individuais. Não há um seqüenciamento explícito para as instruções dentro de um bloco básico. Apenas as dependências de dados dentro de um bloco atômico são representadas pelos campos de destino e fonte das instruções do bloco. Não há qualquer limitação quanto a ordem em que estes blocos são armazenados na memória.

Os resultados desta proposta são melhor apresentados no artigo de Hao, Patt e outros [62], onde é dada ênfase ao mecanismo de aumento do tamanho dos blocos, como meio de aumentar a largura de banda para busca das instruções. Os resultados mostram um aumento no desempenho 12,3% em média para os programas do SPECint95 e um aumento do tamanho médio dos blocos de 5,2 para 8,2 instruções.

Um trabalho muito interessante é o de Vajapeyam [63], que como nesta tese, também se preocupa em otimizar os demais estágios do *pipeline*, além da busca de instruções. O autor apresenta mecanismos para otimizar os estágios de renomeação e despacho, quando um grande número de instruções é executado em paralelo. Os seguintes pontos foram abordados na proposta: (i) particionamento da janela de instruções em vários blocos, cada um contendo um seqüência dinâmica de instruções; (ii) particionamento lógico do banco de registradores em um banco global e diversos bancos locais, com os últimos contendo registradores locais a uma seqüência dinâmica de código; (iii) o registro dinâmico e reutilização da informação de renomeação dos registradores locais a uma seqüência de código dinâmica.

Particularmente, o artigo observa que a largura de banda da renomeação deve crescer proporcionalmente com a largura de banda para busca de instruções. Mesmo com a proposta de métodos parciais de paralelização para a verificação de dependências [37], o estágio de renomeação é limitado pelo número de portas de

leitura necessárias à tabela de mapeamento de registradores, que é proporcional ao número de operandos que está sendo renomeado simultaneamente.

Finalmente, no trabalho de Palacharla [37], os atrasos das principais estruturas nos processadores super escalares são estudados. Primeiro, um arquitetura super escalar genérica é definida. Então, áreas específicas como renomeação de registradores, lógica de ativação e seleção da janela de instruções, e de *bypass* de operandos são analisadas. Cada uma destas estruturas é modelada e simulada com o simulador elétrico *Spice* para tamanhos mínimos de dispositivo de 0,8  $\mu\text{m}$ , 0,35  $\mu\text{m}$  e 0,18  $\mu\text{m}$ .

Os resultados obtidos indicam que para o despacho amplo de instruções, os atrasos aumentam proporcionalmente com a largura de despacho. Para a renomeação de registradores, os experimentos sugerem o uso de técnicas de *pipeline* para particionar tanto a lógica de detecção de dependências como a de acesso à tabela mapeamento através dos diversos estágios do *pipeline*. Para a lógica de ativação e seleção da janela de instruções, não seria possível tal solução, pois esta é uma operação atômica. O *bypass* de operandos também possui as mesmas características. Então, alguma forma de agrupamento das unidades funcionais se faz necessária para diminuir a lógica envolvida.

Para a melhor compreensão desta tese, alguns dos trabalhos acima relacionados, são descritos em detalhes a seguir, iniciando pelo trabalho histórico de Tomasulo.

## 2.2 O Algoritmo de Tomasulo

### 2.2.1 Introdução

O algoritmo de Tomasulo [38], desenvolvido para o sistema IBM 360/91 em 1967, ainda é o mais eficiente para o despacho seqüencial de instruções. Mesmo após o transcurso de três décadas, diversos processadores [39, 40, 41] da atualidade utilizam variações deste algoritmo para o despacho de múltiplas instruções por ciclo.

Também conhecido como algoritmo associativo, o algoritmo de Tomasulo realiza a difusão dos resultados produzidos pelas unidades funcionais através de um barramento global (CDB) que interconecta os componentes do sistema: unidades funcionais, *estações de reserva* e banco de registradores.

A seguir descrevemos a implementação das estações de reserva no processador

### 2.2.2 As Estações de Reserva

As estações de reserva são *buffers* colocados entre o estágio de despacho e o de execução. Elas permitem o despacho de instruções mesmo que os operandos necessários para a sua execução ainda não estejam disponíveis.

No processador IBM 360/91 as instruções utilizadas são do tipo:

$$R_j \text{ op } R_i \rightarrow R_j$$

Os registradores  $R_i$  e  $R_j$  são os operandos fontes da operação  $op$  e  $R_j$  atua também como destino.

A unidade de ponto flutuante desse processador é constituída por um somador e por uma unidade para multiplicação/divisão, como ilustrado na Figura 2.1.

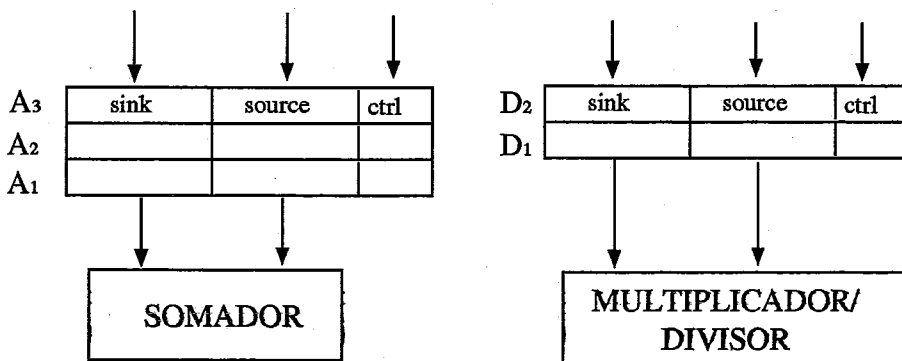


Figura 2.1: Unidades Funcionais e Estações de Reserva do IBM 360/91

A Figura 2.1 apresenta as unidades funcionais e as estações de reserva do IBM 360/91. Na Figura 2.1 as estações de reserva do somador estão identificadas pelos *tags*  $A_1$ ,  $A_2$  e  $A_3$ , e as do dispositivo de multiplicação/divisão pelos *tags*  $D_1$  e  $D_2$ .

Cada estação é formada por cinco campos: dois campos de *tag*, um campo denotando o registrador que atuará como fonte e destino da operação realizada pela unidade funcional (campo *sink*); pelo campo que identifica o registrador como fonte (*src*); e por um campo de controle (*ctrl*).

As instruções de ponto flutuante são despachadas para as estações de reserva mesmo que os seus operandos não tenham sido avaliados. Nesse caso, ao invés de transferir o valor do operando, o algoritmo associativo transfere para a estação

de reserva o *tag* da estação que armazena a instrução que produzirá o resultado esperado. Uma vez despachada, a instrução permanece na estação de reserva até que seja concluída. Nesse momento, ela é descartada, liberando a estação.

A instrução será iniciada quando seus operandos e a unidade funcional apropriada estiverem disponíveis. Graças ao esquema de despacho para as estações de reserva, instruções no IBM 360/91 podem ser iniciadas, executadas e terminadas fora da ordem original em que aparecem no código objeto.

### 2.2.3 O Banco de Registradores

Cada entrada no banco de registradores do IBM 360/91 é formada por três campos. O primeiro campo, denominado *busy bit*, indica se o valor armazenado está atualizado. Caso o valor do registrador ainda esteja sendo produzido, o segundo campo, denominado *tag*, aponta para a estação de reserva contendo a instrução mais recente que especificou esse registrador como destino da operação. Finalmente, o terceiro campo é o registrador propriamente dito. A Figura 2.2 apresenta a organização do banco de registradores do IBM 360/91.

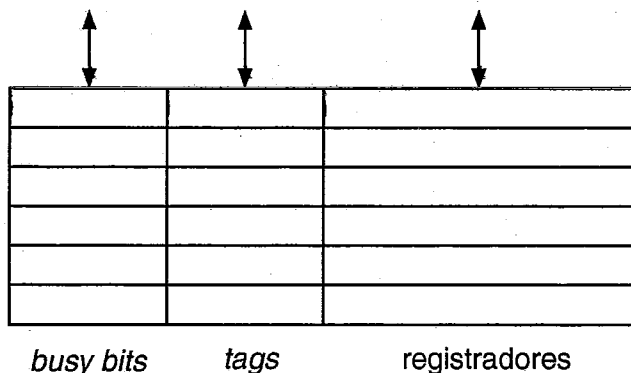


Figura 2.2: Banco de Registradores do IBM 360/91

Estaremos denominando *tag array* [34] o conjunto dos campos de *tags* do banco de registradores. O *tag array* é o responsável pelo fornecimento do *tag* no lugar do operando, quando este não estiver disponível.

O algoritmo de despacho realiza três atividades:

- Antes de transferir a instrução corrente para a estação de reserva, o algoritmo de Tomasulo examina os *busy bits* dos registradores fonte. Se estiveram

atualizados, o conteúdo dos registradores é transferido para os campos *sink* e *src* da estação de reserva. Se o conteúdo de algum desses registradores ainda estiver sendo avaliado, o campo *tag* do registrador fonte é copiado no campo *tag* da estação de reserva onde a instrução atual está sendo alocada;

- Em seguida, o *busy bit* do registrador destino (i.e., o *busy bit* do registrador *sink*) da instrução corrente é atualizado, indicando para as instruções subseqüentes que o conteúdo daquele registrador não está atualizado.
- Finalmente, o algoritmo associativo transfere para o campo *tag* do registrador *sink* a identificação da estação de reserva que recebeu a instrução corrente.

#### 2.2.4 A Difusão de Resultados no CDB

Toda vez que uma unidade funcional conclui uma operação, o resultado é difundido, juntamente com o seu *tag*, através de um barramento global, para o banco de registradores e estações de reserva. O barramento global é denominado CDB (*Common Data Bus*) e a Figura 2.3 apresenta como ele é conectado aos componentes do IBM 360/91.

A difusão é associativa: o resultado da operação é transmitido associativamente para os demais componentes, junto com o *tag* que identifica a estação de reserva que armazena a instrução que produziu o dado. Por exemplo, vamos supor que uma instrução de adição armazenada na estação  $A_2$  tenha sido concluída. Nesse caso, o CDB propaga o resultado da soma e o *tag*  $A_2$  através do CDB.

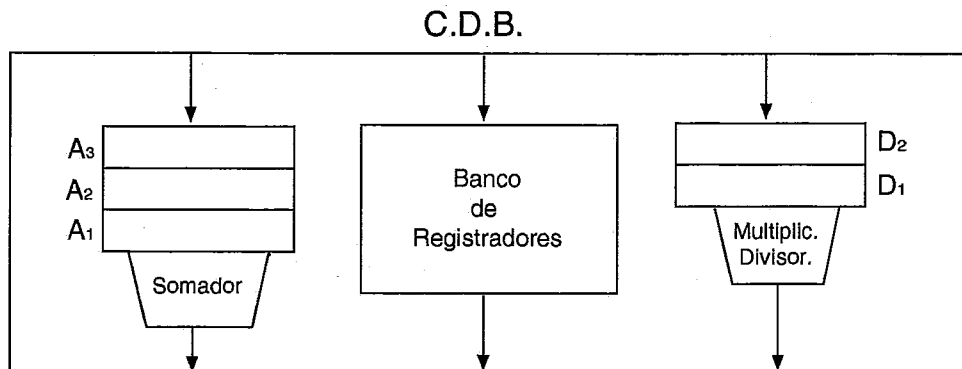


Figura 2.3: Barramento Global do IBM 360/61

Os componentes do sistema monitoram continuamente o barramento de resultados, verificando se o *tag* ora transmitido (e.g.,  $A_2$ ) coincide com o campo de *tag* aguardado. Ocorrendo a coincidência, o resultado é copiado no campo correspondente da estação de reserva ou do banco de registradores. Caso contrário é ignorado. Essa transferência associativa deu origem ao termo “algoritmo de despacho associativo”.

## 2.2.5 Tratamento das Dependências de Dados

Quando da ocorrência de dependências falsas (i.e., anti-dependência e dependências de saída), o algoritmo de Tomasulo simplesmente as ignora, despachando a instrução para uma estação. Se os recursos (unidade funcional e operandos fonte) estiverem disponíveis, a instrução é iniciada imediatamente, antes mesmo do término de suas sucessoras. Independentemente da instrução com dependência falsa ser iniciada ou não, o algoritmo Tomasulo continua despachando as instruções subseqüentes.

No caso de instruções com dependências verdadeiras (i.e., o resultado que ainda será produzido por uma instrução e que será usado como operando fonte por uma outra que a sucede) o algoritmo associativo despacha a sucessora para uma estação de reserva, passa o *tag* discriminando dentre as instruções precedentes que encontram-se nas estações, aquela cujo resultado será o operando fonte. Em seguida, o algoritmo de despacho prossegue, atendendo a próxima instrução.

No IBM 360/91, nem sempre os resultados produzidos pelas unidades funcionais são transferidos para os registradores. Ao simular automaticamente um número infinito de registradores virtuais para a renomeação, o algoritmo de Tomasulo torna desnecessárias as atualizações dos registradores destino de instruções apresentando dependências de saída e anti-dependências. O exemplo a seguir ilustra este fato.

No trecho de programa a seguir vamos supor que **A**, **B**, **C**, **D** e **E** sejam registradores de uma arquitetura implementando o algoritmo de despacho associativo e que *i*, *j* e *k* sejam os endereços identificando as seguintes instruções:

```
i:    A:= B*C;
...   ...
j:    D:= A + 2;
...   ...
```



k:      A := E - 1;

...      ...

As instruções do trecho acima apresentam três tipos de dependências. A instrução  $j$  somente pode ser iniciada após o término da instrução  $i$ . Por outro lado, as instruções  $i$  e  $k$  especificam um mesmo registrador como destino (registrador **A**). Adicionalmente, as instruções  $j$  e  $k$  são anti-dependentes, pois a instrução  $k$  destrói o conteúdo do registrador **A** que é usado como fonte por uma predecessora (a instrução  $j$ ).

Após o despacho da instrução  $i$ , o algoritmo continua atendendo as instruções subseqüentes e quando da chegada da instrução  $j$ , o *tag* identificando a estação de reserva armazenando a instrução  $i$  (i.e., a instrução que produzirá o operando **A**) é copiado no campo apropriado da estação que recebeu  $j$ .

É importante observar que nesse momento o registrador **A** ainda está aguardando pelo resultado que será produzido pela unidade de multiplicação, uma operação que requer vários ciclos de processador para ser concluída. Por esse motivo, a instrução  $j$  permanecerá bloqueada na estação de reserva, aguardando pelo valor **A** para ser iniciada.

Ocorre que o algoritmo de Tomasulo não é interrompido e quando do despacho da instrução  $k$ , o campo de *tag* associado ao registrador **A** será alterado, passando a apontar para a estação de reserva destinada a instrução  $k$ . Ou seja, quando da propagação do resultado produzido pelo multiplicador, o registrador não será atualizado já que ele estará aguardando pelo resultado da instrução  $k$ . Em outras palavras, a operação de atualização foi cancelada pelo algoritmo.

Considerando que a instrução  $j$  ainda mantém o *tag* apontando para o multiplicador, então ela irá utilizar como operando fonte o produto (ao invés de  $E + 1$ ), garantindo desse modo o resultado esperado, conforme especificado pelo programador. Deve-se enfatizar também que o início antecipado de uma instrução provoca um aumento no nível de utilização dos recursos e conseqüentemente reduz o tempo de processamento do programa de aplicação. No exemplo, as instruções sucessoras de  $k$  poderiam ser iniciadas antes mesmo da conclusão da instrução  $j$  e suas predecessoras.

## 2.2.6 Estudo de Caso

Nesta seção apresentamos uma descrição do processador comercial PowerPC 604 [39], que emprega uma versão modificada do algoritmo de Tomasulo para despachar até quatro instruções por ciclo.

A arquitetura PowerPC 604 possui 6 unidades funcionais, cada uma com duas estações de reserva. Além disto, faz uso de uma estrutura denominada *banco de renomeação* e de um *reorder buffer* com 16 entradas, para permitir a execução especulativa e fora-de-ordem.

O banco de renomeação da arquitetura PowerPC evita a contenção nos acessos ao banco de registradores durante a execução fora-de-ordem e, junto com o *reorder buffer*, garantem uma interrupção precisa. Os resultados das instruções são armazenados nos registradores do banco de renomeação e lá permanecem até serem copiados (na ordem especificada pelo programa) nos registradores arquiteturais pelo estágio de conclusão.

A arquitetura do PowerPC 604 possui 32 registradores inteiros e 64 de ponto-flutuante, além dos 28 registradores do banco de renomeação, sendo doze para uso geral (inteiros), oito para ponto flutuante e oito para armazenar os códigos de condição (*flags*).

Uma entrada do banco de renomeação é alocada quando uma instrução que modifica um registrador é despachada. Esta entrada é marcada como alocada, mas não válida. Após a unidade funcional ter executado a instrução, o resultado é escrito no registrador do banco de renomeação e a entrada marcada como válida. Quando a instrução for concluída, o seu resultado é copiado do registrador do banco de renomeação para o registrador arquitetural e a entrada é liberada.

Quando uma instrução é despachada, seus operandos fontes são procurados simultaneamente no banco de registradores arquiteturais e no banco de renomeação. Se um valor for encontrado no banco de renomeação, esse valor é utilizado; caso contrário, o valor é lido do banco de registradores arquiteturais. Note que no banco de renomeação está embutida uma tabela de renomeação com busca associativa, ou seja, são necessários 8 comparadores (um para cada operando lido) para cada registrador do banco de renomeação.

Contudo, pode ocorrer que a entrada no banco de renomeação ainda não esteja

válida, se a instrução correspondente ainda estiver executando. Nesse caso, a instrução é despachada para a estação de reserva *com o identificador da entrada do banco de renomeação* ao invés do operando original. O operando será transferido para a estação de reserva quando o resultado for produzido, através do barramento de resultados.

No algoritmo de Tomasulo, diferentemente, o que é enviado é o *identificador da estação de reserva para onde a instrução que vai produzir o resultado foi despachada*.

Na arquitetura do PowerPC uma instrução não pode ser despachada a menos que todas as instruções precedentes no *buffer* de despacho tenham sido despachadas e apenas uma instrução pode ser despachada para a *mesma unidade funcional a cada ciclo*. O despacho na arquitetura PowerPC é em ordem e, embora possa terminar a execução de até quatro instruções por ciclo, apenas dois resultados podem ser transferidos em paralelo para o banco de registradores arquitetural.

No PowerPC, as estruturas mais solicitadas, em relação ao número de instruções despachadas em paralelo, são principalmente a tabela de renomeação, o banco de renomeação e o banco de registradores arquiteturais.

## 2.3 Predição de Desvios

### 2.3.1 Correlação Dinâmica de Desvio baseada em Caminhos

Nesta seção analisamos uma alternativa aos métodos de predição de desvios que utilizam correlação dinâmica [43]. O artigo aqui analisado, de autoria de R. Nair [55], apresenta um esquema que utiliza o *caminho* que leva até um desvio condicional para prever o desvio mais acuradamente.

Embora não seja um esquema de predição de múltiplos desvios, foi incluído em nosso estudo, por ser utilizado posteriormente em esquemas de predição de múltiplos desvios e de *traces*.

Para a análise do método proposto o autor utilizou a simulação baseada em *traces*. Os *traces* gerados não incluem o código executado pelo Sistema Operacional. Para manter os *traces* com tamanhos razoáveis (50 a 300 milhões de instruções), utilizou-se um subconjunto ou uma versão modificada da entrada padrão de cada programa de avaliação (SPEC92 e IDRAW).

O autor define *slack* de uma instrução de desvio condicional como sendo o número de instruções entre esta e a última instrução da qual ela depende. Ele também definiu o termo “latência do código de condição” de um processador como o número mínimo de instruções que devem existir entre a avaliação do código de condição do desvio condicional e um desvio dependente, de modo que não haja paradas no *pipeline*, devido à execução do desvio condicional. Em uma máquina com *pipeline* simples, esta é a latência do *pipeline* desde o início do estágio de busca até o início do estágio de execução. Por exemplo, se a latência para um determinado processador é 3 e existir um número menor,  $i$  de instruções entre a condição e o desvio, haverá uma penalidade de  $3 - i$  ciclos para o preenchimento do *pipeline* se a predição for incorreta.

O estudo realizou a medida das penalidades associadas às predições incorretas, ao invés de apenas medir as taxas de predição incorretas. Ele definiu esta penalidade com sendo  $k - i$ , onde  $k$  é um parâmetro dependente de implementação que expressa a latência do código de condição, e sendo  $i$  o *slack* para aquela instrução de desvio. Sendo  $B_i$  o número de desvios dinâmicos na aplicação com *slack*  $i$ , e  $N$  o número total de instruções na aplicação, a penalidade para predições incorretas para a aplicação é definida como:

$$\frac{\sum_{i < k} (k - i) B_i}{N}$$

Ao contrário da taxa de falhas, a penalidade de predição incorreta depende tanto da porção de instruções que são desvios condicionais, como também da habilidade do compilador em ordenar os desvios adequadamente.

O modelo proposto na realidade é um modelo simplista, mas acredita o autor que a natureza qualitativa dos resultados não é afetada se simplesmente contarmos o número de instruções entre a condição e o desvio para estimar o *slack*. No artigo é apresentada a adaptação deste modelo quando houver vários *pipelines* na máquina.

Um esquema do método proposto pode ser visto na Figura 2.4. O endereço alvo da instrução de desvio é utilizado como índice de uma Tabela de Histórico de Caminhos, que gera um dos índices para acessar uma Tabela de Preditores. Cada linha desta tabela contém várias predições para um mesmo histórico. A predição correta vai ser selecionada utilizando-se o endereço da instrução de desvio atual.

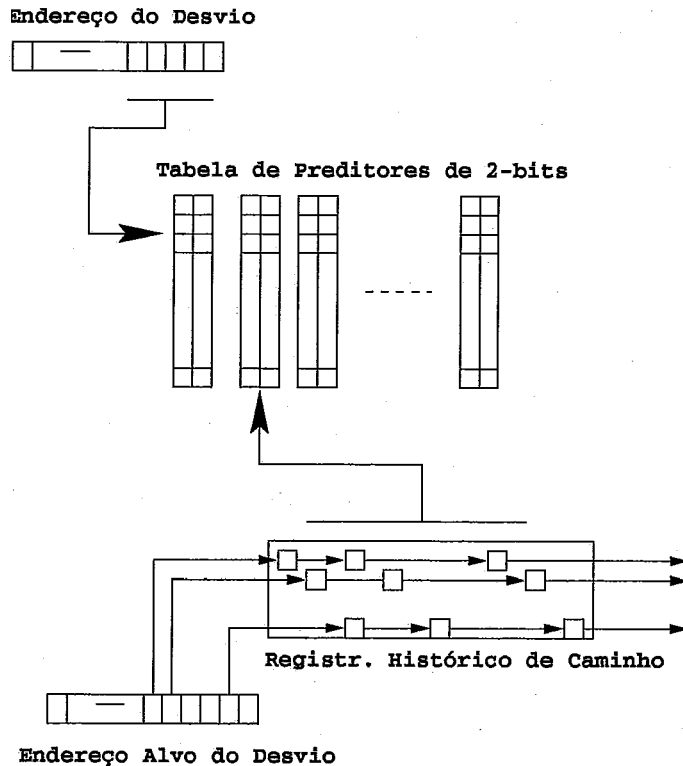


Figura 2.4: Organização de Um Esquema de Correlação por Caminho

O autor apresenta as seguintes conclusões:

- Uma medida importante de como um esquema de predição de desvio se comporta é o grau de degradação do desempenho de uma aplicação quando a predição falha. O esquema de avaliação  $k - i$  provê uma melhor indicação das implicações no desempenho do que a métrica tradicional, isto é, a taxa de falhas da predição de desvio.
- É necessário distinguir o desempenho assintótico dos esquemas de predição de desvio quando utilizado em situações reais. O desempenho real de um determinado esquema para uma dada aplicação depende da quantidade de *hardware* e da habilidade de reter informações “aprendidas” ao longo das trocas de contexto. Algoritmos de predição que se comportam bem apenas assintoticamente precisam de mais *hardware* e dependem do alto reuso da informação “aprendida”. Então, em situações onde o número de aplicações utilizadas simultaneamente é alto, ou quando um grande número de instruções precisa ser executado antes das predições se tornarem acuradas, estes algoritmos podem

sofrer perda de desempenho devido aos problemas de *aliasing* ou à sobrecarga de aprendizado.

- O autor apresentou um esquema com um *hardware* simples que utiliza a informação do caminho executado para correlação de desvio, ao invés de esquemas padrões de resultados de desvios anteriores. O autor mostra que, para um mesmo número de ítems, o histórico de caminho oferece uma melhor predição quando comparado ao histórico de padrões. Contudo, algumas vantagens são mascaradas pelo fato de que mais *bits* são necessários para representar o histórico de caminho do que o histórico de padrões. O desempenho dos esquemas de correlação de padrões e de caminho são equivalentes para um *hardware* equivalente.
- Para tabelas de histórico de tamanho fixo, a escolha de esquemas de predição de desvio depende da frequência esperada com que a informação aprendida é perdida, seja pela troca de contexto ou por *aliasing* de endereços (usualmente resultado de um grande número de aplicações simultâneas). Esquemas que dependam menos no histórico são preferíveis quando esta frequência é alta. Para esquemas não adaptativos, correlação baseada em caminhos com menos ítems de histórico, provê um bom compromisso entre aprendizado rápido e bom desempenho.

### 2.3.2 Predição de Múltiplos Desvios

A predição de múltiplos desvios é importante quando se despacham um grande número de instruções em paralelo. Nesta situação existem sempre blocos básicos distintos sendo buscadas em um mesmo ciclo e uma predição acurada se faz necessária.

O artigo de autoria de Yeh et al. [11] apresenta um mecanismo de predição de múltiplos desvios e busca simultânea de múltiplos blocos básicos não consecutivos.

O mecanismo de predição de múltiplos desvios proposto é uma modificação de um preditor adaptativo de dois níveis de desvios simples [43], que utiliza uma *Cache* de Endereços de Desvio (*Branch Address Cache - BAC*) para fornecer a predição dos endereços de desvio.

O preditor usado como referência foi o preditor adaptativo de dois níveis com uso de um “Registrador de Histórico de Desvio” (*Branch History Register - BHR*) global e de uma “Tabela de Histórico de Padrões” (*Pattern History Table - PHT*) com contadores de 2 *bits*.

Na Figura 2.5 é mostrado um esquema do mecanismo proposto para prever até 3 desvios em cada ciclo. Os  $k$  primeiros *bits* do Registrador de Histórico são utilizados para indexar a Tabela de Histórico e obter a predição para o primeiro desvio. Os  $k-1$  *bits* mais à direita são usados para indexar novamente a Tabela de Histórico para obter a predição para o segundo desvio, sendo que são geradas duas predições: uma TAKEN e outra NOT TAKEN, uma delas será escolhida, dependendo do resultado da predição do primeiro desvio. Já para o terceiro desvio são utilizados os  $k-2$  *bits* mais à direita e são lidas 4 posições na Tabela de Histórico, sendo escolhida uma delas dependendo dos resultados das predições dos desvios anteriores.

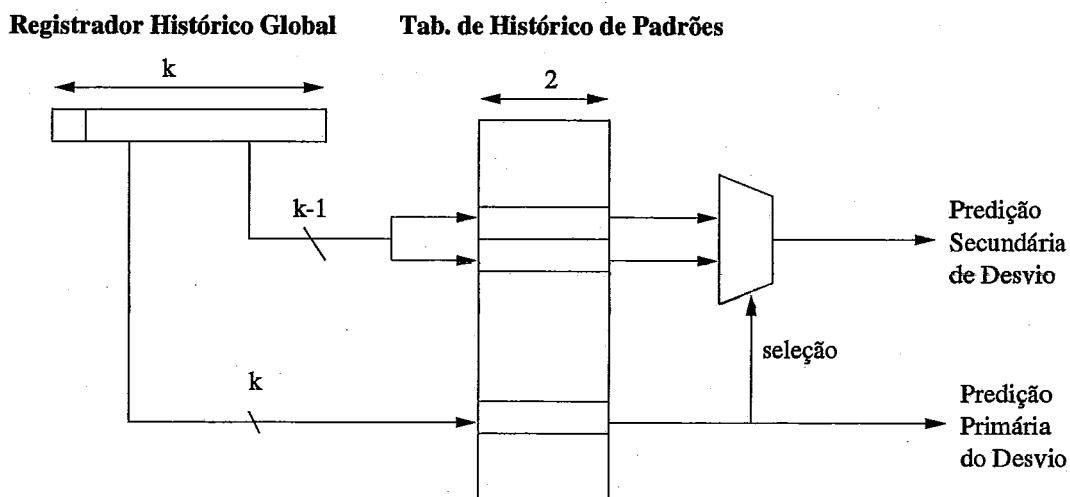


Figura 2.5: Predição de Dois Desvios com apenas uma PHT

Toda vez que é feito um acesso à *Cache* de Instruções, é feito um acesso à *Cache* de Endereços de Desvio, para verificar se há alguma instrução de desvio entre as instruções que estão sendo buscadas. Ao mesmo tempo que é feita a predição de desvio (acesso a PHT) e a *cache* de instruções é lida, um acesso é feito à *Cache* de Endereços de Desvio (BAC), e os *bits* da predição são utilizados para escolher os endereços de busca do próximo ciclo. Se for houver um *hit*, estes endereços são utilizados. Caso contrário, a busca de instruções prossegue sequencialmente.

Nem sempre podem ser realizadas múltiplas predições em todos os ciclos. Esta limitação ocorre nos seguintes casos: quando um bloco básico é muito grande; quando o desvio for uma instrução de retorno; se o desvio for indireto, então o mecanismo de busca das instruções deve aguardar até que os operandos estejam disponíveis.

Apesar da possibilidade da busca simultânea de 2 ou 3 endereços da *cache* de instruções, apenas o endereço do desvio mais “recente” é utilizado para a predição. A *Cache* de Endereços de Desvio guarda todos os endereços cuja escolha seja possível, desse modo, se são previstos até 3 desvios por ciclo, 14 possíveis endereços são armazenados em cada entrada da *Cache* de Endereços de Desvio.

Uma *cache* de desvio suportando duas previsões por ciclo precisaria de 212 *bits* por entrada. Para três previsões por ciclo, seriam 464 *bits* por entrada. Supondo-se uma *cache* com associatividade 4 e 512 entradas, precisamos de um total de 53 Kbytes e 116 Kbytes, respectivamente.

A busca de múltiplos blocos pode ser feita com o uso de uma *cache* de instruções organizada em *interleaving*, que permite obter uma largura de banda suficiente para a leitura de mais de um bloco básico ao mesmo tempo.

Os autores simularam diversas configurações de *cache*. A mais utilizada foi a de uma *cache* de instruções com 32 Kbytes, com associatividade 2, com 8 bancos entrelaçados com portas simples, cada banco com linhas de 16 *bytes* (4 instruções). Cada endereço de busca pode ler dois bancos (duas linhas), garantindo a busca de 5 a 8 instruções por ciclo (a perda é devida ao alinhamento dos blocos básicos). Se houver **conflito**, apenas o primeiro dos acessos será atendido. Nas simulações realizadas, o número máximo de instruções fornecidas ao processador foi de 16 instruções por ciclo.

Na análise do desempenho da proposta foram utilizados programas do SPEC89, inteiros e de ponto flutuante, compilados para o Motorola 88100. Somente os 50 milhões iniciais das instruções executadas de cada programa foram empregadas na avaliação.

O processador não é simulado completamente, ou seja, as únicas fontes de parada (*stall*) são as falhas na *cache* de instruções, predições incorretas de desvio e falhas na *cache* de endereços de desvios nos desvios tomados. A penalidade assumida para um desvio predito incorretamente foi de 6 ciclos. A penalidade para uma falha na



*cache* de instruções considerada foi de 10 ciclos.

A unidade utilizada para a avaliação dos esquemas propostos é o IPC<sub>f</sub>, ou seja, o número de instruções efetivas buscadas por ciclo pelo mecanismo de busca de instruções.

Os resultados apresentados mostram um IPC<sub>f</sub> de 3,0 e 6,6 (para os programas inteiros e de ponto flutuante) com apenas uma predição, de 4,2 e 7,1 para duas predições por ciclo e de 4,9 e 8,9 para três predições por ciclo.

As taxas de acerto para as predições de desvio ficaram entre 91,5% e 98,4% para um BHR com 14 *bits* e entre 93,5% e 98,7% para 16 *bits*. Deve-se notar que os maiores valores foram obtidos para os programas de ponto flutuante. O melhor resultado para um programa inteiro foi o do *espresso*, com cerca de 96%.

Além da taxa de acerto das predições de desvio e do IPC<sub>f</sub>, outros fatores devem ser considerados na avaliação do desempenho. Entre estes, podemos destacar o conflito nos bancos da *cache* de instruções e a própria taxa de falhas da *cache* de instruções. Da avaliação apresentada no artigo, tomamos o programa *gcc* como exemplo. Neste programa a porcentagem de buscas que ocorrem sem nenhum atraso é de 77% , os casos em que há atrasos por predição incorreta são de 9,8%, os conflitos de banco de memória no acesso à *cache* de instruções são cerca de 3,5 % dos casos, e as falhas na *cache* de instruções são de 2,9%. Os atrasos na decodificação, que ocorrem quando uma instrução de desvio não é detectada (“falhas na *cache* de endereços de desvio”), são de cerca de 7,1% dos casos.

Uma crítica que realizamos ao método utilizado é o pequeno número de instruções simuladas, apenas cerca de 0,1% do total de instruções de cada programa, contendo apenas a parte de inicialização de cada aplicativo. Isto compromete a qualidade dos resultados obtidos.

Uma outra consideração a ser feita ao modelo é que, como somente o *front end* do processador é simulado, não é possível obter uma avaliação da influência do esquema proposto sobre o desempenho final da arquitetura, em termos de IPC. Muitas instruções que foram buscadas não poderiam ser despachadas por problemas de dependência de dados. De qualquer modo, as medidas apresentadas são um limite superior de desempenho.

Os resultados da simulação do programa *gcc*, que possui um elevado número de

desvios estáticos, mostram que a predição incorreta de desvios é responsável pelos atrasos em cerca de 10% dos casos. Ou seja, o uso de apenas um BHR global com 14 *bits* não foi suficiente para caracterizar o comportamento do programa quando o número de desvios estáticos é elevado.

A predição simultânea de 3 desvios exige estruturas de armazenamento com 7 portas de leitura, que possuem um alto custo em termos de área e tempo, o que não é colocado explicitamente no artigo.

As falhas na *Cache* de Endereços de Desvio (BAC) também são significativos no programa *gcc*. Acreditamos que o pequeno número de entradas da BAC (512) e o fato de apenas o endereço do primeiro desvio servir como *tag* para a mesma, sejam as principais causas para essas falhas. O uso apenas do primeiro endereço de desvio significa que deve haver também perdas associadas à repetição da mesma informação na BAC. A BAC também tem um custo alto de implementação, pelos diversos campos que possui, notadamente aquelas tabelas para predição de três desvios por ciclo.

## 2.4 *Trace Cache*

De modo a atender às demandas dos processadores com maior largura de despacho, os mecanismos de busca têm que transferir múltiplos blocos básicos a cada ciclo. *Trace cache* é uma estrutura que armazena diversos blocos básicos a cada ciclo, organizando-os de modo que instruções logicamente contíguas fiquem localizados em posições físicas contíguas. Quando um bloco em particular é solicitado, a *trace cache* pode fornecer o bloco solicitado juntamente com outros blocos que o seguiam quando foi executado pela última vez.

Existem duas propostas básicas de implementação para a *trace cache*, uma apresentada por Eric Rotenberg et al [26] e outra por Sanjay Patel et al [54]. A diferença básica entre elas, é que a primeira utiliza uma *trace cache* de pequena capacidade em conjunto com uma *cache* de instruções de grande capacidade (4KB x 128KB), enquanto que a segunda proposta utiliza um esquema inverso (128KB x 4KB). A diferença de desempenho alegada é de cerca de 24% em favor da última.

Apesar destas peculiaridades, as implementações não diferem muito da descrição

a seguir.

A *trace cache* armazena *segmentos* do fluxo dinâmico de instruções, explorando o fato de que muitos desvios são fortemente polarizados em uma direção. Se um bloco básico A é seguido de um bloco básico B o qual, por sua vez, é seguido pelo bloco básico C em um ponto particular na execução do programa, há uma probabilidade que eles sejam executados na mesma ordem em uma próxima vez. Após a primeira execução nesta ordem, eles são armazenados na *trace cache* como uma única entrada. Buscas subseqüentes do bloco A da *trace cache* fornecerão também os blocos básicos B e C.

A Figura 2.6 mostra um esquema do funcionamento da *trace cache*. O endereço do bloco A é apresentado para a *trace cache*. A *trace cache* responde com um *hit* e fornece o segmento solicitado que é constituído dos blocos A, B e C. As estruturas de predição são lidas concorrentemente com a leitura da *trace cache*. Ao final do ciclo, o segmento é comparado com a predição. Como o preditor selecionou B após A, mas D após B, apenas A e B são fornecidos para a execução.

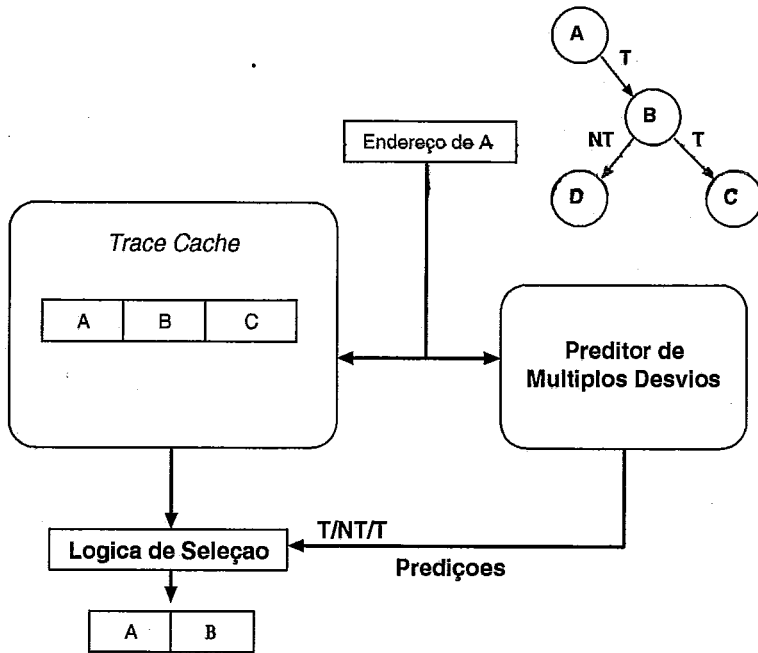


Figura 2.6: Esquema de Funcionamento da *Trace Cache*.

A *trace cache* do exemplo pode armazenar até três blocos básicos por linha. A linha é indexada com o endereço da primeira instrução no primeiro bloco do segmento. A organização da *trace cache* é similar àquela de uma *cache* convencional, onde

as linhas podem ser arrumadas de uma maneira associativa. Um *hit* é determinado por um *tag* idêntico.

Cada linha da *trace cache* do exemplo contém:

- 16 campos para instruções. As instruções são armazenadas já decodificadas e ocupam cerca de 5 *bytes*. Até 3 desvios podem ser armazenados por linha. Cada instrução é marcada com 2 *bits* identificando o bloco básico a que pertence.
- Quatro endereços-alvo. A existência de três blocos básicos por segmento e a possibilidade de buscar segmentos parciais, faz com que haja quatro alvos possíveis para um segmento. Os quatro endereços são armazenados explicitamente permitindo a geração imediata do próximo endereço de busca, mesmo no caso em que apenas uma parte do segmento for utilizada
- Informação do Caminho. Este campo codifica o total e as direções dos desvios no segmento e inclui *bits* para identificar se um segmento termina em um desvio e se aquele desvio é uma instrução de retorno de subrotina. No caso de uma instrução de retorno, uma pilha de endereços de retorno [65] provê o próximo endereço de busca.

O tamanho total de uma linha é cerca de 97 *bytes* para uma arquitetura típica: 5x16 *bytes* para as instruções, 4x4 *bytes* para os *tags* de endereço, e 1 *byte* de informação do caminho.

As dependências entre instruções dentro de um segmento são pré-analisadas antes do segmento ser armazenado na *trace cache*. Os identificadores dos operandos fonte de cada instrução incluem um campo de 2 *bits* indicando se o operando é produzido por uma instrução dentro do segmento ou é produzida por uma instrução despachada em um ciclo anterior. Se o valor for produzido internamente, então o campo indica a qual bloco dentro do segmento pertence a instrução produtora. Com esta informação, a lógica de renomeação pode determinar rapidamente se o *tag* de renomeação para o operando fonte é fornecido pela tabela de renomeação de registradores (RAT - *Register Address Table*) ou pode ser construído sem um acesso à RAT.

O identificador do operando destino é aumentado com um *bit*, indicando se o registrador estará vivo na saída do bloco básico ou não. Os valores que são vivos na saída do bloco são renomeados e recebem um registrador físico. Isto permite que o mecanismo de reparo recupere o estado da máquina depois de predições incorretas de desvio que ocorrem no meio do segmento, sem necessidade de descartar todo o segmento.

Com esta informação adicional, cada segmento que é lido do *trace cache* requer um mínimo de esforço adicional antes de ser inserido na janela de instruções. Somente instruções que tem um operando fonte produzido por uma instrução fora do segmento requerem uma busca na RAT, e apenas instruções que produzem valores que estão vivos na saída do bloco básico requerem um registrador físico. Uma detecção de dependência de instruções complexa não necessita ser feita no segmento fornecido. Sprangle e Patt [60] mostraram que pacotes de instruções que são buscadas e pré-analisadas requerem menos portas de leitura e escrita nas estruturas de renomeação de registradores e no banco de registradores.

Finalmente, segundo Patt, instruções podem ser armazenadas na *trace cache* em uma ordem que permita um despacho rápido, porque as dependências de dados são explicitamente marcadas.

Um segmento só será armazenado na *trace cache* se não houver um *trace* maior já armazenado e que contenha este segmento. Por exemplo, se o segmento *ABC* reside na *trace cache*, um novo segmento *AB* não será adicionado. Contudo, se quisermos adicionar o segmento *ABD*, então *ABC* poderá ser substituído. Para facilitar este trabalho, a informação de caminho é incluída em cada linha da *trace cache*.

Durante o armazenamento na *trace cache*, um segmento é finalizado quando:

1. Contém 16 instruções, ou
2. Contém 3 desvios condicionais, ou
3. Contém um única instrução de desvio indireto, retorno ou *trap*, ou
4. A adição de um novo bloco básico resultará em um segmento com mais de 16 instruções.

A regra 1 é derivada do tamanho da linha da *trace cache* e a regra 2 pelo número de predições fornecidas pelo preditor de desvios. As instruções de retorno e desvios

indiretos causam finalização (regra 3) porque o seu endereço alvo é variável. Desvios incondicionais, contudo, são substituídos por NOPs. Além disto, chamadas de subrotina **não** causam finalização do segmento.

Como os blocos básicos são combinados com o uso de um algoritmo do tipo *guloso*, podem haver casos onde múltiplas cópias do mesmo bloco básico coexistam na *trace cache*. Além disto, os blocos básicos são tratados como unidades atômicas. Um bloco básico não é dividido entre dois segmentos a menos que o próprio bloco básico possua mais que 16 instruções.

Durante o armazenamento na *trace cache*, existem três possibilidades quando da chegada de um novo bloco básico: (1) o novo bloco básico é anexado ao segmento não finalizado criando um segmento maior, mas não é finalizado; (2) o novo bloco básico não pode ser anexado ao segmento não finalizado. No último caso, o segmento é finalizado e um novo segmento não finalizado é criado com o novo bloco básico; (3) O novo bloco básico é completamente anexado ao segmento incompleto e um segmento maior é criado e finalizado.

Finalmente, um novo bloco básico pode ser adicionado a um segmento quando ele for transferido para a janela de instruções ou quando termina a execução. Os compromissos de tal escolha são melhor analisados no artigo de Patt [66].

Para realizar três predições por ciclo, cada entrada da tabela de histórico de padrões (PHT) foi expandida de um único contador de 2 *bits* para 7 contadores de 2 *bits*. A Figura 2.7 mostra como sete contadores são usados para prover três predições por ciclo ( $B_0$ ,  $B_1$  e  $B_2$ ). Os dois primeiros contadores fornecem a predição para o primeiro desvio ( $B_0$ ) e são utilizados para selecionar qual dos dois contadores fornecem a predição para o segundo desvio ( $B_1$ ). Essas duas predições são utilizadas para selecionar um entre quatro contadores para fornecer a predição para o terceiro desvio ( $B_2$ ).

Realizando uma operação de “ou-exclusivo” do endereço de busca atual com histórico global de desvios (BHR) para formar o índice da PHT, como proposto por McFarling [49]. O preditor de desvios possui também uma pilha de endereços de retorno para auxiliar a predição do endereço alvo das instruções de retorno.

Uma *cache* de instruções convencional fornecerá instruções quando a *trace cache* não possuir o segmento requisitado, com uma taxa de até um bloco básico a cada

Saída da Tabela de Historico de Padrões - 64KB

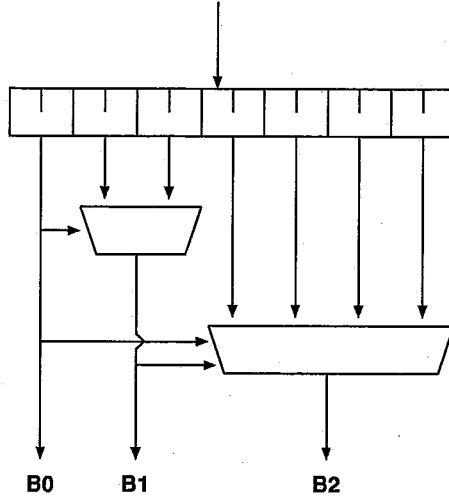


Figura 2.7: Múltiplas Predições por Ciclo

ciclo. Tanto a *cache* de instruções como a *trace cache* são lidas concorrentemente com o mesmo endereço. Como o acesso foi otimizado para o caso em que há um acerto na *trace cache*, haverá o atraso de um ciclo no fornecimento das instruções pela *cache* de instruções, se o acesso falhar na *trace cache*.

O ciclo extra pode ser utilizado para decodificar as instruções buscadas da *cache* de instruções e calcular o próximo endereço de busca. Por esta razão, não é utilizado nenhum *branch target buffer* (BTB) para o mecanismo de busca da *trace cache*.

Apesar de ser uma proposta de muito sucesso e com vários estudos realizados, ainda não há nenhuma implementação de processador comercial que utilize a *trace cache*.

## 2.5 Arquitetura Baseadas em Blocos

### 2.5.1 Renomeação Estático/Dinâmica de Registradores

Em uma arquitetura convencional, uma única instrução é a unidade atômica de trabalho. O trabalho de Sprangle e Patt [60] apresenta um arquitetura onde o bloco básico é a unidade atômica de trabalho.

Os autores defendem que essa arquitetura é mais adequada para um processador super escalar que suporta renomeação de registradores. Primeiro, eles alegam

que podem ser eliminados os  $N \times (N - 1)$  comparadores requeridos pelo estágio de despacho para determinar as dependências entre as  $N$  instruções concorrentes. Segundo, o número de portas do banco de registradores pode ser significativamente reduzido sem perda de desempenho.

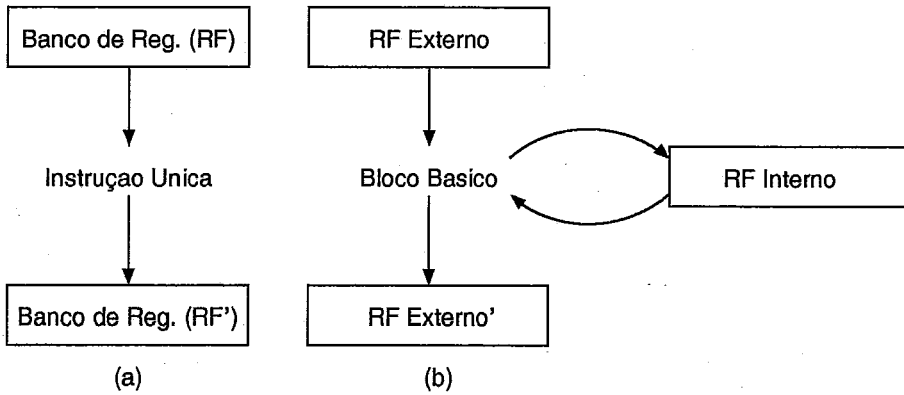


Figura 2.8: Instrução Única vs. Bloco Básico como Unidade Atômica

Numa arquitetura convencional, uma instrução executa a partir de um estado do banco de registradores ( $RF$ ) e gera um novo estado  $RF'$ , como ilustrado na Figura 2.8. A arquitetura proposta pelos autores trata o bloco básico como uma unidade atômica de trabalho. Um bloco básico é executado a partir de um estado do banco de registradores chamado  $RF$  Externo, e gera um novo estado chamado  $RF'$  externo. Instruções individuais dentro de um bloco básico podem utilizar um banco de registradores à parte, chamado  $RF$  Interno. O  $RF$  Interno armazena as instâncias de registradores que são geradas e utilizadas dentro de um bloco básico, enquanto que o  $RF$  Externo armazena as instâncias que são necessárias fora do bloco básico.

A arquitetura associa um “*bit* externo” para cada registrador fonte e destino, como mostrado na Figura 2.9. O “*bit* externo” associado ao operando fonte indica se o dado deve ser lido do  $RF$  Interno ou do  $RF$  Externo. No caso do *bit* associado ao resultado, ele indica se o processador deve escrever os dados apenas no  $RF$  interno ou no  $RF$  Externo também.

Finalmente, a arquitetura proposta introduz a restrição que um registrador em particular pode ser escrito no máximo por uma instrução dentro do bloco básico.



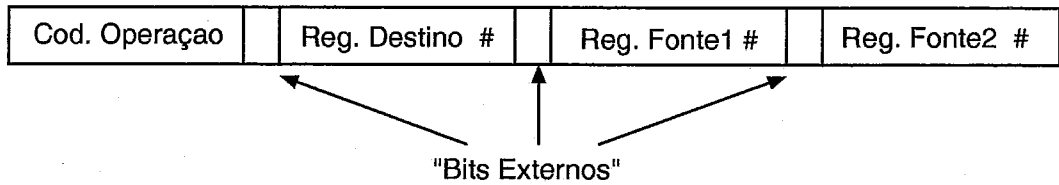


Figura 2.9: Formato de Instrução para Suportar *Tags* Estáticos

Esta restrição simplifica a geração dos *tags* de renomeação, como será explicado adiante. Se um campo de 5 *bits* é usado na representação do registrador, esta restrição limita o tamanho do bloco básico a 32 instruções que podem atualizar registradores. Contudo, a maioria dos blocos básicos possui menos que 32 instruções [67] e, segundo os autores, blocos básicos maiores podem ser divididos com o uso de desvios incondicionais.

A arquitetura proposta elimina a necessidade dos comparadores normalmente necessários para detectar dependências verdadeiras entre instruções despachadas concorrentemente, forçando cada resultado dentro de um mesmo bloco básico a ir para um registrador distinto. Se instruções pertencentes a mais de um bloco básico são despachadas no mesmo ciclo, o *tag* de renomeação pode ser gerado concatenando-se um número gerado dinamicamente para cada bloco básico com o número do registrador gerado estaticamente. Desde que cada registrador de resultado seja único dentro de um bloco básico e um único número de bloco básico seja utilizado para cada bloco básico sendo executado no processador, fica garantido que cada *tag* de renomeação seja único dentro do processador.

A segunda coluna da Tabela 2.1 mostra as instruções como seriam definidas por uma arquitetura com *tags* definidos estaticamente. Se um operando fonte, digamos o registrador  $n$ , se encontra no *RF* Externo, então é marcado como  $(E)R_n$ ; se ele encontra-se no *RF* Interno, então é marcado como  $(I)R_n$ . Se um destino é necessário apenas dentro de um bloco básico, então é escrito apenas no *RF* Interno e marcado como  $(I)R_n$ . Se um destino é "vivo" além do bloco básico, então o resultado deve ser escrito tanto no *RF* Interno como no Externo, e é denotado  $(E'/I)R_n$ .

Quando o estágio de despacho encontra um operando fonte do *RF* Interno, o *RF* Interno é verificado para este valor. Se o valor não foi ainda gerado, então a lógica de despacho forma o *tag* de renomeação concatenando um número único do

Instruções	Como Definido na Arq. Proposta	Como Distribuído para as Estações de Reserva
$R1 \leftarrow R4 + R5$	$(I)R1 \leftarrow (E)R4 + (E)R5$	$(0001)00001 \leftarrow (E)R4 + (E)R5$
$R2 \leftarrow R1 + R6$	$(I)R2 \leftarrow (I)R1 + (E)R6$	$(0001)00010 \leftarrow (0001)00001 + (E)R6$
$R3 \leftarrow R2 * R1$	$(E'/I)R3 \leftarrow (I)R2 * (I)R1$	$(0001)00011 \leftarrow (0001)00010 * (0001)00001$

Tabela 2.1: Despachando Instruções Concorrentemente

bloco básico com o número do registrador. Quando o estágio de despacho encontra um operando fonte de um registrador externo, o  $RF$  Externo é consultado, e lê o valor ou o *tag* do registrador que foi produzido *antes* da execução do bloco básico atual.

A coluna 3 da Tabela 2.1 ilustra as instruções como se fossem distribuídas assumindo o número do bloco básico gerado dinamicamente como (0001). Na mesma tabela,  $(E)R_n$  refere-se ao *tag* ou valor encontrado no  $RF$  Externo quando as instruções são distribuídas, e os operandos de registradores internos são representados com os *tags* apropriados de renomeação.

Uma das limitações impostas pelo modelo proposto é que as instruções devem ser despachadas na ordem inversa, ou seja, “de baixo para cima” como forma de garantir que os resultados não sejam escritos no banco de registradores interno, antes de poderem ser lidos pelas instruções sucessoras e que estão sendo despachadas concorrentemente.

Com um modelo em que um operando fonte é requisitado do banco de registradores externo quando este operando for gerado por uma instrução em um bloco básico anterior, os resultados das simulações demonstraram que é possível uma redução no número total de portas de leitura de 16 para 7, mantendo pelo menos 99% do desempenho de pico.

## 2.5.2 Expansão de Bloco Atômico

Com o objetivo de extrair um maior grau de paralelismo no nível de instrução, os processadores atuais têm sido construídos com maior largura de despacho e com maior número de unidades funcionais. De modo que o potencial de desempenho desses processadores possa ser efetivamente explorado, a taxa de busca das instruções deve ser correspondentemente aumentada.

Neste sentido, uma nova classe de arquitetura é proposta. Este novo tipo de arquitetura foi projetado para endereçar os problemas de desempenho encontrados nos processadores que tentam explorar maiores graus de paralelismo no nível de instrução. Nestas arquiteturas, a unidade atômica arquitetural é o bloco básico. Esta redefinição da unidade atômica permite a simplificação da implementação de muitas tarefas para processadores de despacho amplo.

No artigo aqui analisado [62], Eric Hao, Yale Patt e outros, definem um método de otimização, chamado de “expansão de bloco”, que pode ser aplicado a uma arquitetura estruturada em bloco para aumentar a taxa de busca de instruções do processador.

O bloco básico é denominado de “bloco atômico”, onde apenas as variáveis que são “vivas” na saída do bloco são atribuídas aos registradores arquiteturais. Um resultado não utilizado fora do bloco é referenciado pelo índice da instrução que vai gerar o resultado. Este índice é convertido para um número de registrador físico em tempo de execução.

Por definição não existem anti-dependências ou dependências de saída em um bloco atômico. Não há dependências de saída porque no máximo uma escrita será feita a um mesmo registrador no banco de registradores. Não há anti-dependências, porque sempre o registrador arquitetural será lido, mesmo que esta instrução seja executada depois de uma instrução que altera o mesmo registrador.

A técnica de expansão de blocos combina blocos atômicos separados em um único bloco atômico, aumentando o tamanho médio dos blocos atômicos do programa. Aumentando o tamanho dos blocos atômicos, a taxa de busca de instruções é aumentada sem a necessidade de se buscar múltiplos blocos a cada ciclo.

Os autores alegam que, para os programas inteiros do SPEC95, processadores com arquiteturas estruturadas em bloco executando blocos “expandidos” superam em 12% o desempenho de arquiteturas convencionais.

Quando um bloco atômico é distribuído no processador, ou todas as operações no bloco são executadas, ou nenhuma delas é executada. O uso de uma arquitetura estruturada em blocos simplifica a implementação de um processador de despacho amplo de instruções pela simplificação da lógica necessária para armazenar o estado arquitetural, verificação de dependências, acesso ao banco de registradores e

operações de roteamento para as estações de reserva. Além destes benefícios, as arquiteturas estruturadas em blocos podem aumentar a taxa de busca de instruções do processador com o uso da otimização de “expansão de blocos”.

A expansão de blocos é uma otimização feita pelo compilador que aumenta o tamanho de um bloco básico pela combinação de um bloco com os seus sucessores no fluxo de controle. A Figura 2.10 ilustra como a expansão de blocos funciona. O grafo de fluxo de controle da esquerda consiste dos blocos atômicos *A* – *E*, cada um terminando com um desvio que especifica seus blocos sucessores. O grafo de fluxo de controle a direita mostra o resultado da combinação do bloco atômico *B* com seus sucessores *C* e *D* para formar os blocos expandidos *BC* e *BD*. Ambos blocos *BC* e *BD* são agora sucessores no fluxo de controle ao bloco *A*.

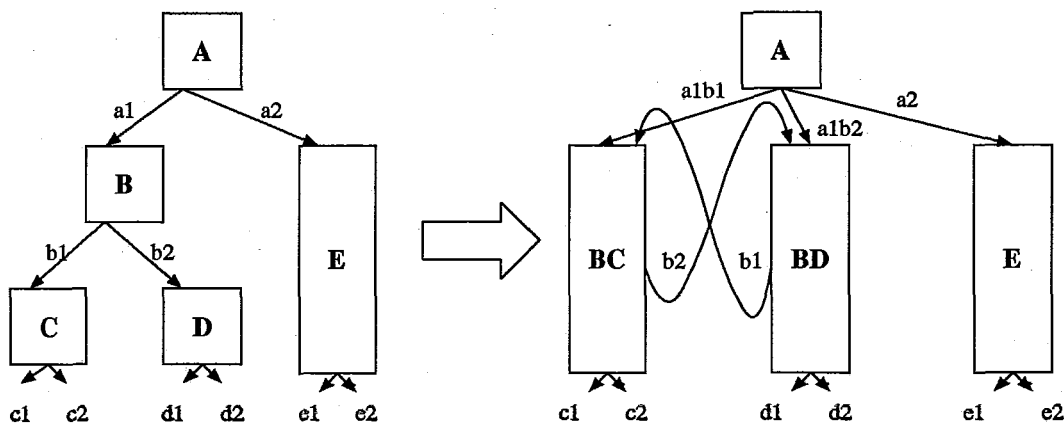


Figura 2.10: Combinando Blocos Atômicos em um Bloco Atômico Ampliado

Como forma de retomar o fluxo correto, no caso de execução do bloco expandido errado, é proposta uma nova classe de instruções de desvio, a instrução *fault*. A instrução *fault* tem como operandos uma condição e um endereço alvo. Se a condição avaliada for falsa, a instrução de *fault* não tem efeito. Se a condição é avaliada como verdadeira, a execução do bloco atômico ao qual pertence a instrução é anulada e o fluxo redirecionado para o endereço alvo. Note que esta definição coloca um caráter especulativo na execução das instruções.

Quando dois blocos são combinados, a instrução de desvio no final do primeiro bloco é transformada em uma instrução de *fault*. O alvo da operação de falha é o bloco expandido que resulta da combinação do primeiro bloco com o seu outro sucessor no fluxo de controle.

Este esquema tem bastante similaridade com o esquema de superbloco proposto por Hwu [68], com a diferença que no esquema de superbloco apenas um dos ramos do desvio é combinado para formar um bloco expandido. Ou seja, a predição de qual dos ramos a ser executado é estática e, usualmente, a predição dinâmica consegue resultados significativamente melhores que a predição estática.

Outro esquema concorrente é a *trace cache* proposta por Jim Smith [69], em que a unidade de busca é composta por duas partes, uma unidade de busca e uma *trace cache*. A unidade de busca traz um bloco básico por vez da *cache* de instrução e a *trace cache* armazena seqüências de blocos básicos executados, combinando-os em um único *trace*. Se o preditor indicar que a seqüência de blocos básicos a serem buscados é a mesma que aquela armazenada na *trace cache*, então o processador pode buscar múltiplos blocos naquele ciclo usando o *trace* da *trace cache*. Este é um meio efetivo de buscar múltiplos blocos a cada ciclo sem os custos associados com outros esquemas baseados em *hardware*.

A diferença fundamental entre a otimização de expansão bloco básico e a *trace cache*, é que a primeira combina seus blocos básicos em tempo de compilação e a última o faz em tempo de execução. Combinando os blocos em tempo de compilação a otimização de expansão de bloco teria como vantagem o uso de toda a *cache* instruções para armazenamento ao invés de um pequeno tamanho que usualmente tem a *trace cache*. O ponto favorável a *trace cache* é que não necessita de mudanças na arquitetura e não aumenta o tamanho do código executável.

Para avaliação dos resultados foram utilizados os programas inteiros do SPEC95 com entradas modificadas de modo que tivessem no máximo 200 milhões de instruções executadas por cada programa. Comparado com uma arquitetura convencional com uma *cache* de 64 Kbytes, com associatividade 4, apresentou uma melhora no desempenho de 12% devido a um aumento no tamanho médio do bloco básico de 5,2 instruções para 8,2 instruções. Para os programas *go* e *gcc*, que possuem um número significativo de blocos básicos, o número de falhas na *cache* de instruções foi maior com uso de executáveis para arquiteturas estruturadas em bloco do que para arquiteturas convencionais. O resultado foi um ganho de apenas 7.2 % para o programa *gcc* e uma perda de 1,5% para o programa *go*.

## 2.6 *Cache* de Renomeações

Os processadores super escalares atuais têm o potencial de buscar múltiplos blocos básicos por ciclo empregando diversos mecanismos de busca [11, 26, 23, 70]. Contudo, este aumento de largura de banda pode não ser aproveitado pelos estágios posteriores do *pipeline* de instruções. No artigo apresentado por Vajapeyam [63], a seguinte análise sobre as dificuldades de renomeação em arquiteturas de ampla largura de despacho foi realizada.

O autor considera que, depois que as instruções são buscadas e decodificadas, os registradores fonte e destino necessitam ser renomeados antes de as instruções serem distribuídas para a janela de instruções, onde aguardam até que os operandos fontes estejam prontos e haja unidades funcionais disponíveis. Em particular, a renomeação de um grande número de instruções por ciclo é difícil. Uma janela de instruções muito grande, necessária para receber múltiplos blocos por ciclo, vai atrasar a resolução de dependências e o despacho das instruções.

O autor alega que o aumento da largura de busca pode resultar nos seguintes gargalos nos estágios subseqüentes do pipeline:

1. A capacidade de renomeação de registradores têm que crescer proporcionalmente com a largura de banda para busca de instruções. O processo de renomeação de registradores têm um componente seqüencial envolvendo a determinação de dependências de dados entre as instruções que estão sendo renomeadas simultaneamente. Mesmo quando métodos parcialmente paralelos sejam utilizados para a verificação de dependências [37], a latência do estágio de renomeação é limitada pelo número de portas de leitura necessárias para a tabela de mapeamento de registradores, que é proporcional ao número de operandos que estão sendo renomeados simultaneamente.
2. A janela de instruções pode crescer demasiadamente se a taxa de despacho não for correspondente a taxa de busca/despacho de instruções. Uma janela com um número maior de instruções pode aumentar o tempo necessário para (i) alimentar um resultado ou *tag* para todas as instruções que estão aguardando na janela de instruções, e (ii) selecionar e despachar  $N$  instruções em um ciclo, entre todas as instruções que estão prontas para execução na janela.

3. A quantidade de registradores físicos necessários para suportar uma janela de instruções muito grande pode aumentar o tempo de acesso aos registradores para mais de um ciclo de relógio. Cada instrução na janela de instruções necessita de um único registrador físico de destino. Então, uma janela de instruções com 256 instruções pode precisar de até 256 registradores físicos.
4. Uma janela de instruções muito grande necessita de um grande número de unidades funcionais para explorar todo o paralelismo disponível. A complexidade de adiantar os resultados das unidades funcionais cresce quadraticamente com o número de unidades funcionais, já que cada saída de uma unidade funcional necessita ser distribuída para as entradas de todas as demais unidades funcionais.

O artigo em questão [63] apresenta as seguintes propostas para enfrentar estes problemas:

- Particionamento da janela de instruções em múltiplos blocos, cada um contendo uma seqüência dinâmica de código;
- Particionamento lógico do banco de registradores em um banco global e diversos bancos locais, os últimos contendo registradores locais a uma determinada seqüência dinâmica de código;
- Registro dinâmico e reuso da informação de renomeação para registradores que são locais a uma seqüência dinâmica de código.

Uma janela de instruções tradicional pode aumentar o ciclo de relógio do processador. A proposta do artigo é a organização da janela de instruções em linhas de *trace*, como proposto na *trace cache* [26]. Uma linha de *trace* se constitui de um número de blocos básicos sucessivos dinamicamente na execução do programa, e é unicamente definido [26] com o endereço da primeira instrução e os resultados de cada um dos desvios nessa seqüência dinâmica.

Cada linha de *trace* buscada da *trace cache* é decodificada, renomeada e despachada para uma única linha de *trace* na janela de instruções, chamada de *janela de trace* na Figura 2.11. Neste exemplo, cada linha de *trace* possui seu conjunto de unidades funcionais. Trabalhos anteriores de [71] mostram que a maior parte do

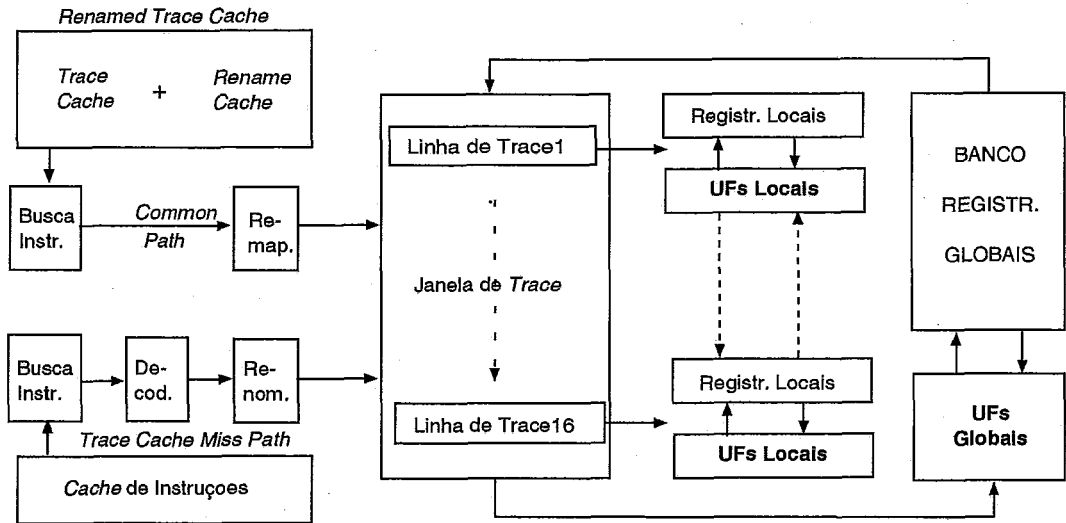


Figura 2.11: Janela de Instruções Organizada em *Trace Lines*

destino das instruções é consumido dentro das próximas 32 instruções dinâmicas depois que foram produzidos. Deste modo, pretende-se reduzir o tráfego de operandos para o banco de registradores global.

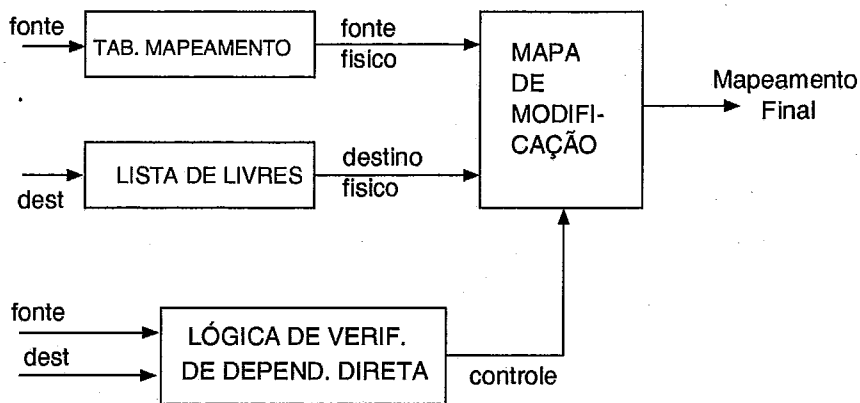
A largura de banda para o despacho das instruções é limitada a um pequeno número de instruções (cerca de duas) por ciclo por linha de *trace*. Segundo os autores, este valor é suficiente para explorar o paralelismo disponível na linha de *trace*, sem aumentar demasiadamente a demanda por área de silício para implementação de caminhos de *by-pass*.

Cada instrução na janela de instruções, exceto desvios e instruções de *store*, necessita de um registrador físico de destino. O uso de janelas de instruções demasiadamente grandes provocam um aumento no número de registradores a serem implementados no banco de registradores.

Contudo, registradores de destino locais a linha de *trace* podem ser renomeados para um banco de registradores físicos locais. Este banco de registradores será liberado quando do término da execução da linha de *trace*.

A renomeação tradicional de registradores envolve a busca de tabela de mapeamento para pegar os mapeamentos atuais para os registradores arquiteturais e, em paralelo, a verificação de dependências verdadeiras, seguida de uma modificação do mapeamento para os registradores fontes que foram produzidos por instruções predecessoras no mesmo grupo de instruções que está sendo renomeado no momento,





Tradicional

Figura 2.12: Renomeação de registradores no Modo Tradicional

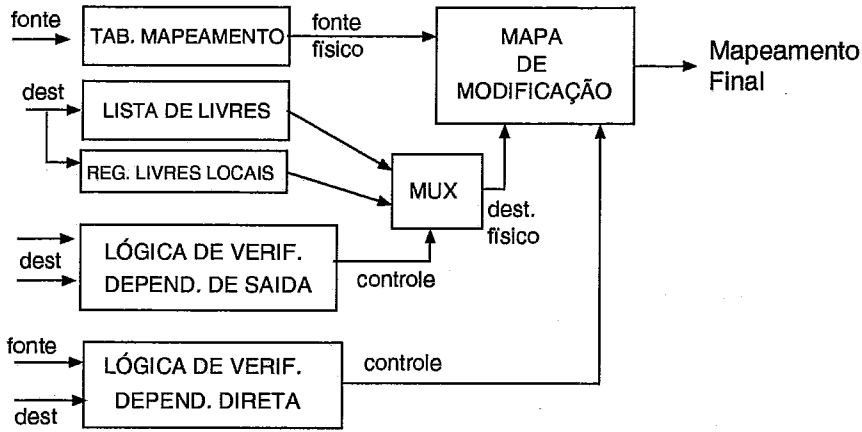
conforme a Figura 2.12.

Para identificar registradores “vivos-na-saída”, a verificação de dependências de saída é feita em paralelo com a verificação de dependências verdadeiras, como visto na Figura 2.13. Em paralelo, um único registrador físico local é atribuído provisoriamente a cada operando de destino. Se a lógica de verificação indica que a saída de uma instrução é reescrita por uma outra instrução posterior, o mapeamento local deste registrador é que é enviado para os estágios posteriores de modificação da tabela de mapeamento.

Como forma de diminuir a pressão sobre a tabela de mapeamento, é sugerido que a renomeação seja feita em grupos de 4. Cada uma linha de *trace* com 16 instruções. Esta opção, contudo, tem um efeito dramático no desempenho do processador e a opção sugerida para contornar este problema é o uso de *renamed-trace cache*.

Nesta *renamed-trace cache*, *bits* adicionais são utilizados para guardar a informação de renomeação, e apenas os registradores “vivos-na-entrada” e “vivos-na-saída” necessitam ser remapeados, de modo que isto pode ser feito em apenas um ciclo de relógio. Uma *trace line* aparece para o estágio de renomeação com o sendo uma única instrução CISC com múltiplos operandos fontes e destinos, e apenas um mapeamento de registradores arquiteturais para físicos é necessário.

A Tabela 2.2 mostra o desempenho da arquitetura proposta, a *trace window*, contra o de uma arquitetura com busca ideal, mostrando que os resultados são



Método c/ Cache de Renomeação

Figura 2.13: Renomeação de Registradores com Uso de *Trace Lines*

bastante promissores. Os programas de avaliação utilizados são do SPECint92.

Máquina	Busca Ideal Máx. ILP 16	<i>Trace Window</i> Máx. ILP 16
cc1	10,80	4,40
compress	7,93	2,25
eqntott	11,31	3,65
espresso	14,96	7,42
xlisp	11,91	6,17

Tabela 2.2: Comparação de Desempenho

Finalmente, o autor coloca que o aumento da largura de despacho ressalta os custos associados com outras fontes de “bolhas” no *pipeline*, tais como erros de predição de desvios e falhas na *cache* de instruções, mostrando a importância de reduzir estes custos.

## 2.7 Registradores Físico-Virtuais

### 2.7.1 Introdução

Nos processadores super escalares, as dependências falsas podem ser eliminadas com o uso da técnica de renomeação dinâmica de registradores. Neste contexto, o nome

pelo qual um registrador é referido pelas instruções é chamado de *registrador lógico ou arquitetural*. A posição que ele ocupa no banco de registradores é denominada de *registrador físico*.

O tempo de acesso ao banco de registradores representa um dos atrasos mais críticos dos microprocessadores atuais e é esperado que se torne mais crítico a medida que os futuros processadores aumentem a largura de despacho e o tamanho da janela de instruções. Banco de registradores com tamanhos maiores e com um maior número de portas têm tempos de acesso maiores [35, 36] que penalizam o desempenho do processador.

Analizamos aqui dois artigos [72, 73], dos mesmos autores, que apresentam uma proposta que permite atrasar a alocação dos registradores físicos até o último ciclo do estágio de execução. Esta proposta, baseada em uma arquitetura com janela de instruções centralizada, foi denominada de “registradores físico-virtuais”.

O procedimento proposto diminui o número de ciclos durante os quais cada registrador físico é alocado. Com isto é possível conseguir economia significativa no número de registradores necessários e, como consequência, reduzir os tempos de acesso ao banco de registradores.

A seguir, realizamos uma descrição mais detalhada desta proposta.

## 2.7.2 Renomeação de Registradores

O objetivo da renomeação de registradores é remover as dependências falsas entre as instruções. Existem duas abordagens principais para prover posições para o armazenamento dos registradores de renomeação:

- Nas entradas do *reorder buffer*. Neste caso, o resultado de cada instrução é guardado no *reorder buffer* até que seja atualizado no registrador arquitetural. Os operandos fonte de uma instrução são lidos ou do banco de registradores ou de uma entrada do *reorder buffer*, quando as instruções são transferidas para as estações de reserva. Os operandos que não estiverem prontos durante a decodificação da instrução serão fornecidos, posteriormente, diretamente das unidades funcionais para as estações de reserva. Quando uma instrução é concluída, seu resultado é copiado do *reorder buffer* para o banco de registradores. Uma variação deste esquema inclui o uso de um banco de renomeação,

que evita o armazenamento dos resultados no *reorder buffer* [39].

- Em um registrador físico. Neste caso o banco de registradores físico contém mais registradores do que aqueles definidos pela arquitetura. Com o uso de uma tabela de renomeação, os registradores arquiteturais são mapeados para os registradores físicos no estágio de decodificação. Neste esquema os operandos são sempre lidos do banco de registradores físico.

Ambos esquemas de renomeação de registradores são utilizados pelos processadores mais recentes. O primeiro é utilizado pelo Pentium Pro [40], PowerPC 604 [39] e SPARC64 [41]. O segundo pelo MIPS R10000 [32] e DEC 21264 [33].

Nos artigos aqui analisados, o segundo esquema é utilizado nas investigações realizadas. Segundo os autores, a vantagem principal da organização dos registradores físico-virtuais, que é a alocação dos registradores físicos para renomeação por um período de tempo mais curto, também existe quando é comparada com o primeiro esquema.

### 2.7.3 Descrição do Esquema

No esquema proposto, quando uma instrução é decodificada, o seu registrador de destino é mapeado para um *tag*. Neste caso, os *tags* não são relacionados com nenhuma posição de armazenamento e serão chamados de *registradores físico-virtuais (VP)*. Mais tarde, quando a instrução terminar a execução, ela aloca um registrador físico para armazenar o resultado. Finalmente, quando a instrução é concluída, o registrador físico alocado pela instrução precedente para o mesmo registrador arquitetural de destino é liberado.

O esquema de registradores físico-virtuais pode ser utilizado tanto para registradores inteiros como de ponto flutuante e inclui o uso de duas tabelas de mapeamento, como mostrado na Figura 2.14. A primeira delas se chama GMT (Tabela Geral de Mapeamento), que é indexada pelo número do registrador arquitetural e contém três campos:

- Registrador VP - Contém o último *tag*, indicando o registrador físico-virtual, para o qual o registrador arquitetural foi mapeado.

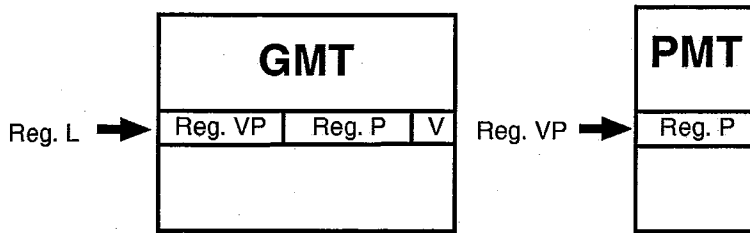


Figura 2.14: Tabelas de Mapeamento

- Registrador P - Indica o último registrador físico para qual o registrador arquitetural foi mapeado.
- *Bit V* - Indica se o campo P contém um valor válido.

A outra tabela é chamada de Tabela de Mapeamento Físico (PMT). Ela tem uma entrada para cada registrador físico-virtual e contém o último registrador físico para o qual este foi mapeado.

Além disto, existe uma lista de registradores físicos livres e outra para os registradores físico-virtuais disponíveis. O número de registradores físico-virtuais deve ser igual ao número de registradores arquiteturais mais o tamanho da janela de instruções.

Para cada nova instrução que é decodificada, os seus operandos fonte são renomeados para um registrador físico-virtual ou para um registrador físico, se houver algum disponível. Para cada registrador fonte, a tabela GMT é indexada. Se o *bit V* estiver setado ('1'), o registrador arquitetural é renomeado para o registrador físico especificado no campo P. Caso contrário, é renomeado para o registrador físico-virtual.

No caso do registrador destino, se houver, será renomeado para um registrador físico-virtual livre. A entrada correspondente na GMT é atualizada, o *V bit* é colocado em '0' e o valor antigo é salvo no *reorder buffer*, para restaurar o contexto em caso de uma predição incorreta ou exceção.

A instrução é então despachada para a fila de instruções, junto com os registradores fonte e destino substituídos pelo mapeamento descrito e com mais dois *bits*, que serão colocados em '1' quando ambos operandos fontes estiverem mapeados para registradores físicos.

A Figura 2.15 mostra a estrutura do *reorder buffer*:

- Registrador L: identificador do registrador arquitetural de destino;
- *Bit* C: um *bit* que indica se a instrução completou sua execução;
- Registrador VP: este campo indica o mapeamento físico-virtual da última instrução que tinha o mesmo registrador arquitetural como destino.

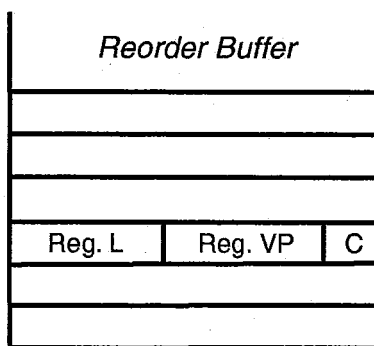


Figura 2.15: *Esquema do Reorder Buffer*

Uma instrução será despachada quando ambos operandos fonte estiverem mapeados para registradores físicos. Os operandos são lidos e a instrução é executada.

Cada instrução cujo operando destino for um registrador aloca um registrador físico no último ciclo de sua execução. Um novo registrador físico é alocado de uma lista de livres. A PMT é então atualizada para refletir o novo mapeamento físico-virtual para físico.

Simultaneamente, o identificador físico-virtual para o operando destino é propagado para todas as entradas da fila de instrução junto com o identificador do novo registrador físico alocado (colocamos aqui uma observação sobre o custo desta operação). Estes dois valores também são propagados para a GMT. Cada entrada compara seu identificador de registrador VP com aquele propagado e, no caso de coincidência, o identificador do registrador físico é copiado no campo registrador físico P e o campo V é colocado em '1'. Finalmente, o campo C da entrada correspondente do *reorder buffer* é colocado em '1'.

Quando uma instrução é concluída, o registrador físico-virtual alocado pela instrução anterior (e que está identificado pelo campo VP no *reorder buffer*) é liberado.

O registrador físico alocado pela mesma instrução anterior também é liberado. O identificador desta instrução é obtido pelo acesso à PMT, indexando-a com o valor do registrador VP que está sendo liberado.

No caso de uma exceção ou desvio predito incorretamente, as entradas do *reorder buffer* são retiradas uma a uma, da mais nova até a que ocasionou o erro. Esta operação desfaz os mapeamentos realizados, mas deve-se notar que é complexa e de alto custo.

O esquema proposto, contudo, pode ser vítima de uma situação de *deadlock* que iremos detalhar a seguir.

#### 2.7.4 Evitando o *Deadlock*

Pode acontecer que uma instrução termine a execução e não haja nenhum registrador físico disponível. Uma solução imediatista seria fazer com que esta instrução aguardasse até que um registrador físico fosse liberado.

Contudo, se a instrução em questão for a instrução mais velha da janela, as demais instruções ficam impedidas de atualizarem os resultados no registrador arquitetural e nenhum registrador físico será liberado, o que resultaria em um *deadlock*.

No primeiro artigo analisado [72] é proposta uma solução, considerada ineficiente e logo abandonada em favor de uma nova solução proposta em um artigo posterior [73], descrita a seguir.

Cada instrução aloca um registrador no último ciclo do estágio de execução se houver registradores físicos livres. Se uma instrução alcançar o último ciclo do estágio de execução e não houver registradores físicos disponíveis, é verificado se existe alguma outra instrução mais nova que já tenha alocado um registrador. Neste caso, é melhor roubar o registrador alocado pela instrução mais nova e atribuí-lo à instrução mais velha.

Em seguida são tecidas várias considerações sobre a eficiência do método, comparado com um esquema convencional de renomeação. Alegam os autores que a reexecução das instruções é vantajosa, pois permite validar previamente a predição de desvios; com a execução prévia de instruções de *load*, é possível antecipar a busca de dados da memória principal para a *cache* de instruções.

Finalmente, os autores argumentam que os resultados produzidos pelas ins-

truções, poderiam ser guardados em um *buffer* para serem utilizados por um mecanismo de reuso de instruções. Quando os registradores físicos fossem liberados, os resultados poderiam ser copiados diretamente para os novos registradores físicos. Entretanto, não é feita uma análise mais aprofundada deste tema no artigo apresentado.

## 2.7.5 Resultados

Os resultados obtidos da simulação dos programas do SPEC95 com um modelo convencional e um modelo com a organização de registradores físico-virtuais.

A organização de físico-virtual apresenta um *speed-up* médio, medido em termos de IPC, de 5% e 24% para o códigos inteiro e de ponto flutuante, respectivamente.

A percentagem de instruções que tem seu registrador físico roubado, é de 5,26% para os programas inteiros e de 11,76% para os programas de ponto flutuante. Isto resulta em uma re-execução de 9,79% e 57,75% das instruções respectivamente.

O artigo apresenta outros dados sobre a vantagem da re-execução prévia de instruções. As instruções de *load*, por exemplo, experimentam um razoável grau de *prefetch*, em média 28%.

## 2.8 Resumo

Realizamos neste capítulo uma revisão da literatura relacionada com o tema, com destaque para os trabalhos mais importantes.

Procuramos mostrar a diversidade dos problemas que envolvem a busca e execução de múltiplas instruções por ciclo, mostrando o estado da arte nas soluções propostas até o momento.

Neste sentido, existe um grande número de trabalhos voltados para a predição de desvios e para a busca eficiente de instruções. Outros pontos importantes se referem à pressão que o grande número de instruções exerce sobre os estágios posteriores de pipeline.

Optamos por concentrar nossos estudos na otimização dos estágios de despacho e conclusão, onde encontramos diversas deficiências nos modelos atuais, para o despacho de múltiplas instruções.



# Capítulo 3

## Os Modelos Propostos

### 3.1 Introdução

Existem dois modelos principais de arquiteturas super escalares implementados em processadores comerciais: com janela de instruções centralizada e com janela de instruções distribuídas (algoritmo de Tomasulo). Exemplos de processadores que utilizam o primeiro modelo são o MIPS R10000 [32] e Alpha 21264 [33]. Os processadores PowerPC 604 [39], Pentium Pro [40] e HAL SPARC64 [41] estão entre aqueles que empregam o segundo.

Os dois modelos apresentam as deficiências já relacionadas no Capítulo 1 e 2, quando do despacho amplo de instruções (mais do que 4 instruções em paralelo). Dentre estas deficiências destacamos o grande número de comparadores necessários para a detecção das dependências verdadeiras entre as instruções que estão no estágio de despacho ( $O(n^2)$ ) e o grande número de portas de leitura/escrita em dispositivos de armazenamento como banco de registradores e tabela de renomeação.

Neste capítulo apresentamos um modelo de arquitetura que utiliza o algoritmo de Tomasulo modificado, para o despacho mais eficiente de múltiplas instruções por ciclo. O modelo básico representa uma família de processadores super escalares e é caracterizado:

- Pelo uso de um *future file* em conjunto com um *reorder buffer*, que implementam o conceito de execução especulativa. O uso do *future file* resulta em um banco de registradores mais simples, tornando mais eficiente a busca de operandos;

- Por uma *cache* de dependências que torna mais eficiente o mecanismo de detecção de dependências verdadeiras quando do despacho de várias instruções por ciclo;
- Por de um mecanismo de despacho seletivo, que reduz o número de barramentos de resultado e de comparadores nas estações de reserva. Esta técnica utiliza estações de reserva especializadas, com um número de comparadores proporcional ao número de operandos ainda não resolvidos.

Alguns processadores comerciais [39, 40, 41] adaptaram o algoritmo de Tomasulo para o despacho de múltiplas instruções. Entretanto, como mostraremos na Seção 3.6, estas soluções são deficientes quando se despacha um grande número de instruções em paralelo.

## 3.2 Eliminando As Dependências Falsas

Um grande número de processadores super escalares utilizam uma tabela de renomeação para eliminar as dependências falsas e para implementar a execução especulativa e interrupção precisa. Nesta seção apresentaremos soluções alternativas adotadas no nosso modelo para eliminar as dependências falsas.

O algoritmo de Tomasulo tem soluções bem definidas para a eliminação de dependências falsas. Estas soluções, reconhecidamente eficientes e de baixo custo, permitem executar as instruções fora da ordem especificada pelo programa. Desta forma, aumentam as oportunidades de exploração do paralelismo no nível de instrução.

Entretanto, o algoritmo de Tomasulo foi desenvolvido para despachar apenas uma instrução por ciclo. Adaptações são necessárias para que seja possível despachar mais de uma instrução por ciclo.

No algoritmo de Tomasulo, assim que uma instrução é despachada, as instruções subseqüentes podem modificar os registradores fonte desta instrução, sem provocar anti-dependências, porque os valores destes operandos (ou os *tags* correspondentes) já foram copiados na estação de reserva. As dependências de saída são resolvidas impedindo a escrita de um resultado no banco de registradores, se o *tag* correspondente a este resultado não for o mais recente.

No nosso modelo, para o despacho de múltiplas instruções por ciclo, as anti-dependências são tratadas da mesma forma: copiando o *tag* ou os operandos nas estações de reserva. Ou seja, se entre as instruções que estão sendo despachadas em paralelo houver alguma cujo registrador destino também é fonte de uma instrução precedente, a consistência dos dados é mantida porque o valor do registrador fonte em questão será copiado antes que ele seja alterado.

Um exemplo pode ser visto na Figura 3.1. Estamos supondo que a instrução *j* escreve no registrador r3, que é lido como operando fonte por uma instrução precedente *i*. No ciclo T1 os operandos fonte (r2, r3, r4 e r5) das duas instruções são copiados nas estações de reserva RS1 e RS2. A instrução *j* pode ser executada antes da instrução *i* e alterar o valor do registrador r3, porque o valor original já foi copiado na estação de reserva RS2 e ficará preservado até que instrução *i* seja executada.

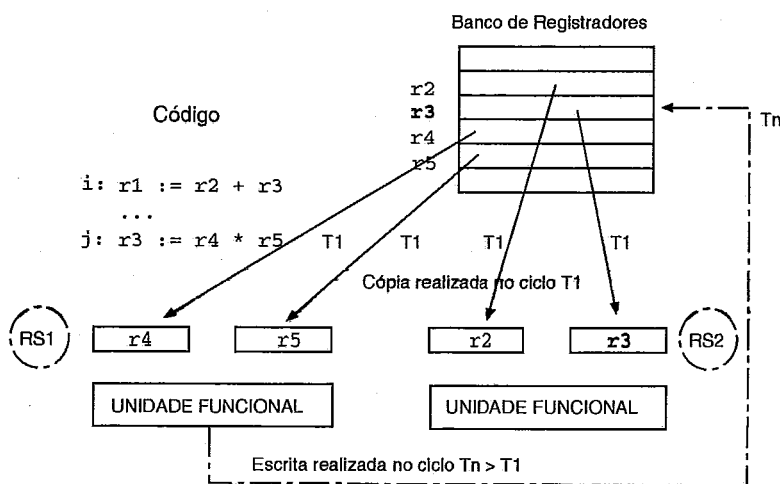


Figura 3.1: Resolução de Anti-Dependência

Quanto às dependências de saída, deve-se garantir que a entrada correspondente no *tag array* do banco de registradores aponte para a estação de reserva que recebeu a instrução mais recente (podemos ter várias instruções despachadas no mesmo ciclo alterando um mesmo registrador).

Nosso modelo inclui um mecanismo garantindo que somente o resultado da instrução mais recente seja transferido para o banco de registradores. O *tag array* possui um esquema de prioridade embutido que, ao detectar duas escritas para o mesmo registrador, dá prioridade para a instrução mais recente. A Figura 3.2 ilustra

esse mecanismo.

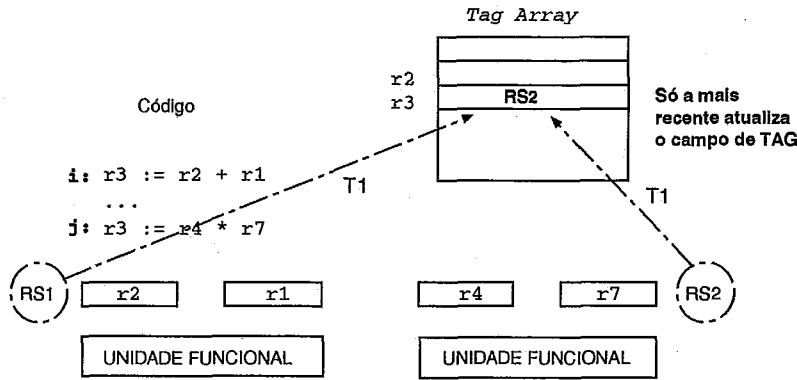


Figura 3.2: Resolução de Dependência de Saída

A Figura 3.2 apresenta um trecho de programa, o *tag array*, as estações de reserva e unidades funcionais.

No ciclo T1, é determinado que as instruções *i* e *j* serão despachadas para as estações de reserva RS1 e RS2, respectivamente. Neste momento, temos que escolher qual das instruções irá alterar o registrador r3, colocando no *tag array* do banco de registradores o identificador da estação de reserva que vai gerar o resultado mais recente. Como o *tag array* é um dispositivo com várias portas de escrita, o decodificador deve assegurar que apenas a instrução *j* atualize a entrada correspondente ao registrador r3 com o valor RS2.

Embora as dependências falsas possam ser resolvidas com os mecanismos aqui apresentados, para o caso da detecção das dependências verdadeiras o procedimento é mais complexo. Todas as instruções despachadas precisam verificar se os seus operandos fonte são produzidos por alguma outra instrução precedente que está sendo despachada simultaneamente.

### 3.3 A Detecção das Dependências Verdadeiras

A detecção das dependências verdadeiras entre as múltiplas instruções que estão sendo despachadas em paralelo constitui-se num grande problema dado a sua complexidade. São necessários  $N \times (N - 1)$  comparadores [34] para determinar as relações de dependências entre as  $N$  instruções que estão sendo despachadas. Para ilustrar

a extensão deste problema, apresentamos o exemplo a seguir.

BB1:

```
7b3e4 sethi %hi(0xfa800), %o0 ! Independente
7b3e8 lduw [%i7 - 0x14], %o1 ! Externa
7b3ec or %g0, %o1, %o2 ! Interna (0x7b3e8)
7b3f0 sll %o2, 2, %o1 ! Interna (0x7b3ec)
7b3f4 or %o0, 0x268, %o0 ! Interna (0x7b3e4)
7b3f8 lduw [%o0 + %o1], %o1 ! Interna (0x7b3f4 + 0x7b3f0)
7b3fc lduw [%i7 - 0x28], %o0 ! Externa
7b400 subcc %o1, %o0, %g0 ! Interna (0x7b3f8 + 0x7b3fc)
7b404 ble BB3 ! Interna (0x7b400)
7b408 nop ! Independente
```

BB2:

```
7b40c sethi %hi(0xf9c00), %o0 ! Independente
7b410 lduw [%o0 + 0x140], %o1 ! Interna (0x7b40c)
7b414 lduw [%i7 - 0x24], %o2 ! Externa
7b418 sub %o1, %o2, %o0 ! Interna (0x7b410 + 0x7b414)
7b41c lduw [%i7 - 0x14], %o1 ! Externa
7b420 add %o1, %o0, %o0 ! Interna (0x7b41c + 0x7b418)
7b424 stw %o0, [%i7 - 0x14] ! Interna (0x7b420) + Externa
```

BB3:

```
7b428 sethi %hi(0xd6000), %o0 ! Independente
7b42c sethi %hi(0xd2000), %o1 ! Independente
7b430 lduw [%i7 - 0x14], %o2 ! Externa
7b434 lduw [%i7 - 0x30], %o3 ! Externa
7b438 add %o2, %o3, %o2 ! Interna (0x7b430 + 0x7b434)
7b43c or %g0, %o2, %o3 ! Interna (0x7b438)
7b440 sll %o3, 2, %o2 ! Interna (0x7b43c)
7b444 or %o1, 0x154, %o1 ! Interna (0x7b42c)
7b448 lduw [%o1 + %o2], %o2 ! Interna (0x7b444 + 0x7b440)
7b44c or %g0, %o2, %o1 ! Interna (0x7b448)
7b450 sll %o1, 2, %o2 ! Interna (0x7b44c)
7b454 or %o0, 0x3a0, %o0 ! Interna (0x7b428)
7b458 lduw [%o0 + %o2], %o1 ! Interna (0x7b454 + 0x7b450)
7b45c lduw [%i7 - 0x2c], %o0 ! Externa
7b460 subcc %o1, %o0, %g0 ! Interna (0x7b458 + 0x7b45c)
7b464 bne BB4 ! Interna (0x7b460)
7b468 nop ! Independente
```

Obs: %g0 é sempre lido como zero.

Independente - Instruções que podem ser despachadas imediatamente.

Externa - Indica dependência com instruções fora do bloco básico.

Interna - Indica dependência com instruções no mesmo bloco básico.

Este exemplo mostra um trecho em linguagem *assembly* de três blocos básicos (BB1, BB2 e BB3) do programa *go*. A primeira coluna indica o endereço da instrução na base hexadecimal e na coluna seguinte é mostrado o mnemônico da instrução seguida de seus operandos. O comentário no final de cada linha indica os tipos de dependências da instrução, conforme especificado na legenda ao final do código de exemplo.

O trecho de programa correspondente possui as características mostradas na Tabela 3.1. Nesta tabela, a segunda coluna indica o total de instruções de cada bloco básico; a coluna seguinte mostra o número de instruções independentes, ou seja, quantas instruções podem ser despachadas imediatamente, por não possuírem dependências verdadeiras; a quarta coluna indica quantas instruções possuem algum tipo de dependência; a coluna seguinte indica quantos **operandos** nas instruções de cada bloco básico possuem dependências verdadeiras com instruções em **outros** blocos básicos que já foram despachados; e na última coluna é listado o número de **operandos** das instruções de cada bloco básico que possuem dependências verdadeiras com instruções no **mesmo** bloco básico.

Bloco Básico	Total de Instruções	Instruções Independentes	Instruções c/ Dependência	Dependência Externa	Dependência Interna
BB1	10	2	8	2	8
BB2	7	1	6	3	6
BB3	17	3	14	3	15

Tabela 3.1: Resumo das Dependências do Trecho do Programa *go*

Note que o número de dependências internas é significativamente maior que o número de dependências externas. Esta característica é convenientemente explorada pelos nossos modelos de *cache* de dependências.

A Figura 3.3 apresenta o fluxo de controle do trecho do programa *go*. Ela mostra três blocos básicos (BB1, BB2 e BB3) e o número de vezes que eles foram ativados durante a execução do programa. A seqüência de blocos básicos {BB1, BB3} foi executada 84.316 vezes e a seqüência {BB1, BB2, BB3} 21.228 vezes.

A Tabela 3.2 mostra o total de comparações necessárias para o despacho das primeiras 16 instruções deste trecho de programa, para diversas larguras de despacho. Em outras palavras, foram consideradas as dez instruções de BB1 e as seis

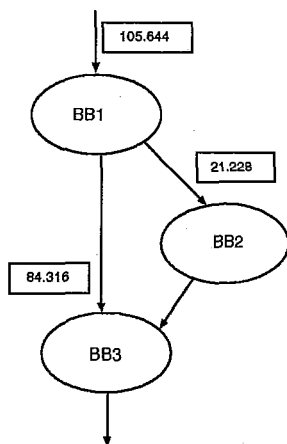


Figura 3.3: Diagrama de Fluxo de Controle dos Blocos Básicos

primeiras de BB2 (ou de BB3). Para uma largura de despacho de 4 instruções, serão necessários quatro ciclos para o despacho destas instruções, para uma largura de 8 instruções, dois ciclos e apenas um ciclo quando a largura de despacho for de 16 instruções.

Largura	Comparações
4	2.469.968
8	6.691.728
16	11.927.664

Tabela 3.2: Total de Comparações para o Despacho

Podemos observar que quanto maior for a largura de despacho, maior será o número de comparações necessárias. Foi para reduzir esse significativo número de comparações que desenvolvemos três modelos de *cache* de dependências, detalhados mais adiante.

### 3.4 Modelo Básico de Arquitetura

Na especificação dos modelos que serão apresentados a seguir, consideramos que derivam do nosso modelo básico: uma arquitetura especializada na execução de blocos básicos [74, 75] que implementa o repertório de instruções da arquitetura SPARC, versão V8. Seu banco de registradores é convencional (i.e., sem o uso de janelas), sendo 32 registradores inteiros e 32 de ponto flutuante.

Este modelo básico é do tipo super escalar com *pipeline* de instruções formado por seis estágios: busca, decodificação, despacho, distribuição, execução e escrita. As máquinas experimentais despacham e executam várias instruções por ciclo; nesta tese estaremos investigando máquinas capazes de despachar até 8 ou 16 instruções por ciclo.

O algoritmo de Tomasulo foi modificado para despachar múltiplas instruções por ciclo, usa um *future file*, um *reorder buffer*, múltiplos CDB's e um banco de registradores com *tag array* que inclui múltiplos comparadores. As motivações para esta escolha serão detalhadas na seção seguinte.

O diagrama do *pipeline* que estaremos utilizando pode ser visto na Figura 3.4.

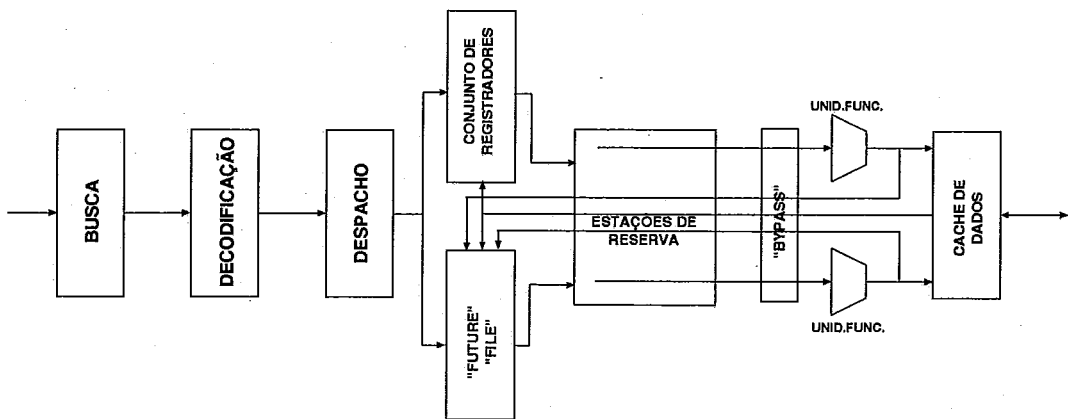


Figura 3.4: Modelo da Arquitetura Super Escalar Básica

As instruções são buscadas na *cache* de instruções, conforme indicado pelo mecanismo de predição de múltiplos blocos básicos. Nos experimentos realizados, consideramos um preditor de desvios onisciente. A cada ciclo é buscado um número suficiente de blocos básicos para preencher a largura de despacho da arquitetura. Existem memórias *cache* separadas para dados e instruções com 100% de taxa acerto.

Um bloco básico pode estar armazenado integralmente no *buffer* de instruções ou não (vide Figura 3.5). Se somente uma parte do bloco básico estiver armazenado, então teremos duas possibilidades: como ocorre com o bloco "d" da Figura 3.5 (b), as instruções iniciais do bloco básico são armazenadas nas últimas posições do *buffer*; ou as instruções finais do bloco básico são armazenadas no início do *buffer*, conforme ocorre com o bloco "c" na Figura 3.5 (b).



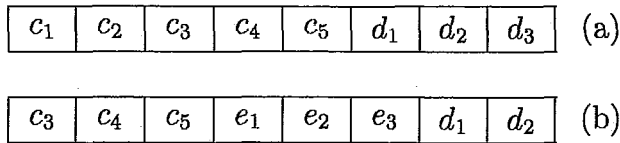


Figura 3.5: Armazenamento de Blocos Básicos no *Buffer* de Despacho

A Figura 3.5 apresenta dois instantâneos de um *buffer* de despacho que é capaz de armazenar oito instruções. Os blocos básicos “*c*,” “*d*” e “*e*” são formados por 5, 3 e 3 instruções, respectivamente. O *buffer* de despacho (a) armazena integralmente os blocos básicos “*c*” e “*d*.” No segundo instantâneo, as três últimas instruções do bloco “*c*” estão armazenadas nas posições iniciais do *buffer* (b) enquanto que o bloco básico “*e*” está carregado integralmente. Somente as duas primeiras instruções de “*d*” estão armazenadas, pois a capacidade de armazenamento do *buffer* (b) foi esgotada. Por esse motivo, apenas no próximo ciclo é que as instruções restantes do bloco “*d*” serão armazenadas e processadas.

Os endereços dos blocos básicos e os índices  $i$  de cada instrução dentro do respectivo bloco básico são armazenados e enviados para os estágios posteriores da máquina. Por exemplo, no *buffer* de despacho (a), as instruções do bloco básico  $d$  serão enviados com os índices 1, 2 e 3. Estes índices serão utilizados pelo mecanismo de detecção de dependências.

A seguir, as instruções são enviadas para o estágio de decodificação, onde é verificado quais os operandos que são necessários, se o código de condição será utilizado e qual o tipo de unidade funcional requerido pela instrução (unidade inteira ou de ponto flutuante). As instruções são então transferidas para o estágio de despacho.

No estágio de despacho, são realizadas as comparações para verificar as dependências verdadeiras entre as instruções que estão sendo despachadas. As instruções serão enviadas para as estações de reserva escolhidas a partir de uma lista de livres. Como os recursos são ilimitados, sempre é possível despachar todas as instruções do *buffer* de despacho.

No estágio de distribuição, as instruções já estão nas estações de reserva, junto com os operandos ou *tags* correspondentes, que foram copiados no início deste ciclo. Este estágio verifica quais instruções estão prontas para execução, isto é, com

todos os operandos disponíveis. Apesar do preditor onisciente, os desvios são enviados normalmente para execução nas unidades inteiras após a resolução de suas dependências.

No estágio seguinte as instruções serão executadas, de acordo com a latência de cada unidade funcional. Nossos modelos consideram um número ilimitado de unidades funcionais.

Finalmente, após o término da execução das instruções, os resultados são transmitidos através dos diversos CDB's, também em número ilimitado, propagando-se para as estações de reserva e banco de registradores.

### 3.5 Despacho Seletivo de Instruções

Uma alternativa para implementar os conceitos de interrupção precisa e execução especulativa, consiste em usar um *future file* e um *reorder buffer* [42]. Para a largura de despacho utilizada nos processadores atuais (4 instruções), o uso desta alternativa implicaria na duplicação dos custos, sem nenhuma vantagem em termos de desempenho [34]. Por esse motivo, o uso do banco de renomeação foi a opção adotada no PowerPC.

Contudo, para maiores larguras de despacho e/ou de janela de instruções, o número de registradores de renomeação necessários para manter um bom desempenho aumenta consideravelmente. Para manter o desempenho em 90% daquele obtido com um número ilimitado de registradores, precisamos de um banco com 80 registradores, no caso de uma janela com 32 instruções e largura de despacho de 4 instruções. Para uma janela com 64 instruções e 8 instruções despachadas por ciclo, o tamanho ideal para o banco de registradores passa para 128 registradores [36].

O impacto do tamanho do banco de registradores no desempenho de uma arquitetura é muito grande: o tempo de acesso depende do número de registradores e do número de operações de escrita e leitura em paralelo. Um modelo analítico [35] que desenvolvemos fornece o tempo de acesso e área gastos para implementar um banco de registradores permitindo múltiplos acessos. A Figura 3.6 apresenta o impacto do tamanho e do número de portas do banco de registradores no tempo de acesso, com tecnologia de  $0.8\mu\text{m}$ .

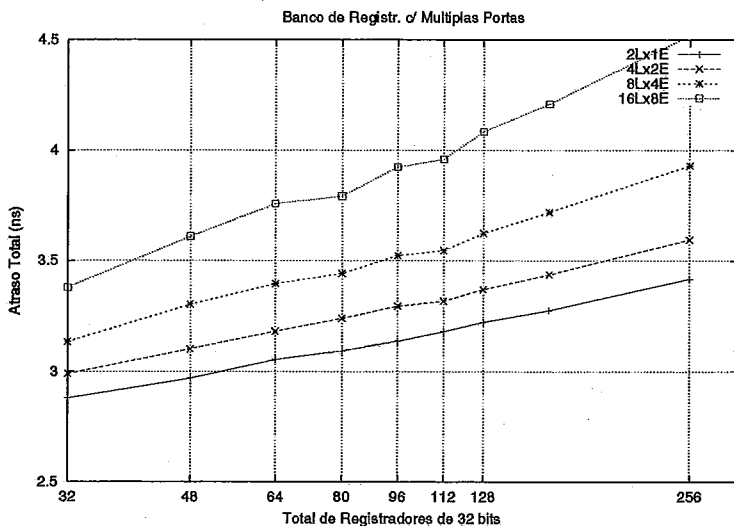


Figura 3.6: Tempo de Acesso ao Banco de Registradores

Ou seja, com o aumento da largura de despacho, o esquema que usa *future file* e um *reorder buffer* passa a ser vantajoso. Este esquema foi adotado no nosso modelo básico.

Para viabilizar a execução especulativa é necessário mover o *tag array* para o *future file*. O campo de *tag* é aumentado de um *bit* para indicar se é o banco de registradores ou o *future file* que armazena o valor mais atualizado. Quando uma instrução é despachada, o valor do operando ou o *tag* é lido do *future file*.

Uma entrada no *reorder buffer* é reservada durante o despacho de uma instrução. Se o resultado de uma instrução de desvio condicional que alcançou o topo for diferente do previsto, o mecanismo de execução especulativa simplesmente descarta as instruções subseqüentes, restaura os registradores do *future file* (com o conteúdo dos registradores arquiteturais) e reinicia a execução a partir das instruções pertencentes ao caminho correto do desvio.

Nosso modelo dispensa a tabela de renomeação. Porém, como mostraremos a seguir, o *tag array* possui complexidade similar à tabela de renomeação.

No esquema convencional, a tabela de renomeação necessita de tantas portas de leitura quantos forem os operandos lidos pelas instruções no estágio de decodificação. Esta tabela também precisa de uma porta de escrita para cada instrução que está sendo despachada. O uso de *shift registers* nas células de memória da tabela de renomeação, aumenta ainda mais a complexidade da tabela de renomeação. Estes

*shift registers* são utilizados para armazenar mapeamentos intermediários (*checkpoints*), resultantes da execução especulativa de desvios. Se a predição de algum desvio estiver errada, o *shift register* é deslocado no sentido inverso até obter-se o último mapeamento correto.

O modelo que desenvolvemos para o banco de registradores [35] também pode ser aplicado à tabela de renomeação, mostrando que os tempos de acesso se tornam mais críticos à medida que aumenta o número de portas.

O equivalente em nosso modelo à tabela de renomeação é o *tag array* [34]. O *tag array* fornece o *tag* no lugar do operando, quando este não estiver disponível. Nessa estrutura, o número de portas de escrita deve ser equivalente ao número de resultados que serão produzidos pelas instruções que estão no estágio de despacho.

Analogamente, o número de portas de leitura do *tag array* deve ser suficiente para fornecer os *tags*, ao invés dos valores dos operandos fonte para todas as instruções que estão sendo despachadas. O número de portas de leitura deve permitir também a leitura dos *tags* que serão comparados com os *tags* das instruções cujos resultados estão sendo produzidos no estágio de execução. Como no esquema com renomeação, também é preciso que o *tag* seja escolhido de uma lista de estações de reserva “livres”.

No caso de execução especulativa não há necessidade de espaço para armazenar os *checkpoints* no *tag array*, pois todas as instruções após o desvio especulado incorretamente serão eliminadas das estações de reserva, enquanto que as precedentes já atualizaram seus resultados no banco de registradores.

Ou seja, embora a tabela de renomeação tenha sido dispensada, o *tag array* tem uma complexidade equivalente: enquanto o *tag array* possui mais portas de leitura que a primeira, a tabela de renomeação precisa de um espaço adicional para armazenar os *checkpoints*.

Na estrutura proposta para nossa arquitetura, realizamos ainda mais uma adaptação no algoritmo de Tomasulo: a adição de tantos CDBs quantos forem os resultados produzidos a cada ciclo. Para cada CDB adicionado, a lógica de associatividade (responsável pelas comparações que verificam se o resultado difundido é o esperado) tem que ser replicada em todas as estações de reserva [76] e também no *tag array*.

Para aliviar os efeitos deste problema, especializamos as estações de reserva, enviando as instruções para estações de reserva com um número de comparadores

proporcional ao número de *tags* que podem receber. Assim, instruções que não têm operandos de leitura ou que tivessem todos os seus operandos disponíveis poderiam ser enviadas para estações de reserva sem comparadores. Este é o mecanismo que denominamos “despacho seletivo”.

Nesta nova técnica de despacho, as instruções são despachadas para as estações de reserva de acordo com o número de operandos que ainda não estão prontos. As estações seriam especializadas, com capacidade diferenciada para aguardar por operandos **não** prontos. Cada uma delas poderia aguardar por um número diferente de operandos, variando entre 0 e 3. Desta forma, um menor número de comparadores e caminhos de dados seria necessário para a implementação do conjunto de estações de reserva.

### 3.6 *Cache* de Dependências

Durante a execução de um programa, as relações de dependências internas, isto é, as relações entre as instruções de um mesmo bloco básico, são imutáveis. Os registradores arquiteturais atualizados por cada bloco básico também sempre serão os mesmos.

Em outras palavras, toda vez que um mesmo bloco básico for ativado, as dependências entre suas instruções serão sempre as mesmas. O mesmo nem sempre ocorre com as instruções de blocos básicos distintos, pois a ordem de execução dos blocos básicos pode ser alterada dinamicamente.

Caso duas ou mais instruções de um mesmo bloco básico atualizem um mesmo registrador, apenas o resultado da instrução mais recente é que precisa ser propagado para o registrador arquitetural, ficando assim disponível para instruções em outros blocos básicos.

Estas características motivaram o desenvolvimento de três modelos de *cache* de dependências, cujo objetivo é reduzir o número de comparações necessárias para determinar as dependências verdadeiras entre as instruções que são despachadas em um mesmo ciclo.

A *cache* de dependências é uma estrutura de *hardware* que armazena as relações de dependências entre as instruções de um mesmo bloco básico. Há diversas pos-

sibilidades para coletar e armazenar as relações de dependências. Uma delas seria através de ferramentas estáticas, para análise do código objeto, em tempo de compilação. Uma outra forma seria incluir mecanismos de controle em estágios anteriores ao despacho, como por exemplo no estágio que realiza a busca de instruções da memória principal para a *cache* de instruções. Finalmente, uma terceira opção seria realizar a coleta durante o próprio estágio de despacho, neste caso implementado segundo um *pipeline*. A escolha final dependerá muito das características de cada projeto, e os prós e contras de cada opção não foram analisados nesta tese.

Os três modelos de *cache* de dependências desenvolvidos são: simples (CS), inteligente (CI) e avançada (CA). A complexidade da *cache* de dependências varia conforme o modelo, assim como os benefícios que podem ser obtidos com o seu uso. Uma arquitetura básica (DE), usada como referência, é compartilhada pelos três modelos de *cache* de dependências e modificações são adicionadas à *cache* simples, de modo a obtermos as *caches* inteligente e avançada.

### 3.6.1 Descrição do Buffer de Despacho

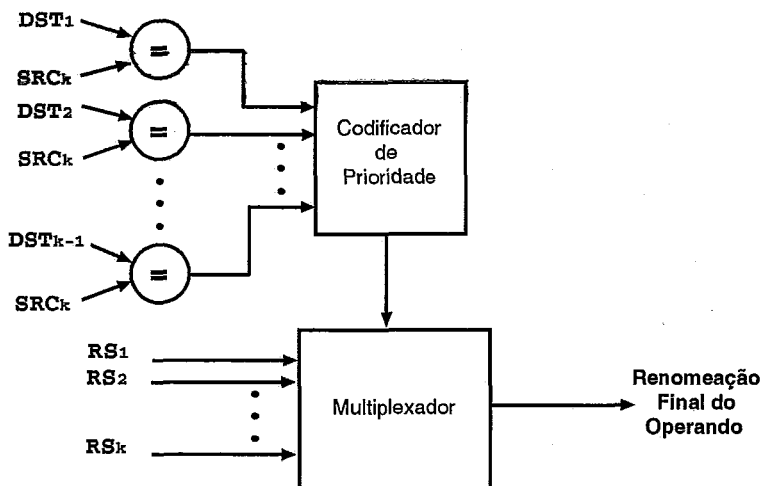


Figura 3.7: Verificação de Dependências no Despacho

O *buffer* de despacho possui características especiais para permitir a sua utilização com a *cache* de dependências. A complexidade do estágio de despacho varia de acordo com o modelo de *cache* de dependências, mas algumas características básicas são compartilhadas entre os diversos modelos.

Na arquitetura básica (DE) é necessário comparar cada uma das instruções com todas as instruções precedentes do *buffer* de despacho. Se alguma dependência verdadeira for detectada, o registrador fonte da instrução deve ser substituído pelo *tag* da estação de reserva que vai armazenar a instrução origem da dependência. O circuito da Figura 3.7 realizaria esta função para cada instrução que está sendo despachada.

Nesta figura,  $SRC_k$  é um dos operandos da instrução na  $k$ -ésima posição do *buffer* de despacho. Ele será comparado com cada um dos operandos destino de cada uma das instruções precedentes no *buffer* de despacho, identificados por  $DST_j$ , onde  $j$  varia de 1 até  $k - 1$ . Um codificador de prioridades detecta a instrução mais próxima em que houve coincidência. O *tag* ( $RS_i$ ) da estação de reserva escolhida para esta instrução é utilizado como *tag* do operando fonte da instrução  $k$ .

Quando utilizado com um dos modelos de *cache* de dependências (CS, CI e CA), é preciso manter um rígido controle sobre os blocos básicos e instruções que forem armazenadas no *buffer* de despacho. Normalmente, mais de um bloco básico pode ocupar o *buffer* de despacho.

Para manter este controle, o *buffer* de despacho recebe as informações sobre os blocos básicos que foram transferidos e, para cada uma das instruções, os correspondentes índices das instruções que indicam a sua posição dentro do bloco básico. Concatenando-se o endereço do bloco básico com o índice de cada instrução, a informação de dependência pode ser extraída da *cache* de dependências.

A nossa arquitetura orientada a blocos básicos, que tem a informação do tamanho de cada bloco básico, facilita estas operações de controle do *buffer* de despacho. Entretanto, mesmo para arquiteturas convencionais, os modelos de *cache* de dependências podem ser utilizados, à custa de um *hardware* de controle adicional.

Para os modelos de *cache* de dependências apresentados, o *buffer* de despacho deverá contar com multiplexadores que permitam compartilhar os comparadores existentes para todas as instruções no *buffer* de despacho. O número e a organização destes multiplexadores varia de acordo com o modelo de *cache* de dependências implementado. Um esquema simplificado para o despacho de uma instrução pode ser visto na Figura 3.8.

Os componentes são basicamente os mesmos utilizados na arquitetura básica, com exceção da inclusão dos multiplexadores, necessários para que diversas ins-

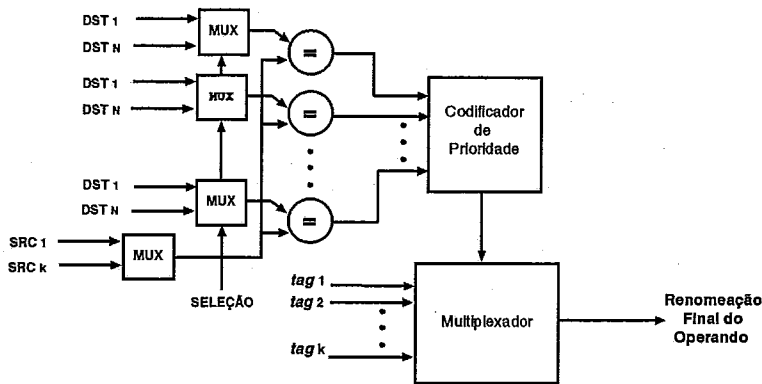


Figura 3.8: Modificação para uso da *Cache* de Dependências

truções possam compartilhar um mesmo comparador. Este esquema introduz algum atraso e obteremos ganho somente se a redução do número de comparadores necessários para o despacho de cada instrução for significativa.

### 3.6.2 *Cache* de Dependências Simples

A *cache* simples armazena as relações de dependências entre as instruções de um bloco básico (denominadas dependências internas).

A sua estrutura é composta por uma memória *cache* com  $N$  entradas, indexadas pelos  $\log_2(N + 2)$  bits menos significativos do endereço do bloco básico, onde  $N$  é uma potência de dois. Estaremos utilizando uma *cache* com mapeamento direto e cada entrada possui um *tag* contendo os bits mais significativos do endereço do bloco básico com  $32 - \log_2(N + 2)$  bits.

Um diagrama de cada entrada da *cache* de dependências é apresentado na Figura 3.9.

Tag de Endereço	Tamanho	Identificador de Dependências		
$32 - \log_2(N + 2)$ bits	4 bits	4 bits	4 bits	4 bits
		Op. Fonte 1	Op. Fonte 2	Cód. de Condição

Figura 3.9: Controle das Dependências Internas

Cada entrada é formada pelo *tag* de endereço, um campo de 4 bits com o total de instruções do bloco básico e até 16 campos de identificação de dependências, um campo para cada instrução do bloco básico. Cada campo de identificação possui



12 *bits* que apontam para as instruções precedentes (pertencentes ao mesmo bloco básico) que irão produzir os operandos fonte requeridos por cada instrução.

Cada campo de identificação de dependências possui 3 grupos de 4 *bits* que indicam, respectivamente, qual é a instrução dentro do bloco básico que gera o primeiro e o segundo operando fonte, além do código de condição. Em arquiteturas sem o código de condição, este último grupo pode ser suprimido. Se nenhuma instrução dentro do mesmo bloco básico produzir o operando, o grupo de 4 *bits* correspondente será preenchido com "0".

Durante a execução de um programa, o *buffer* de despacho é preenchido com oito ou 16 instruções a cada ciclo. Para cada instrução desse *buffer*, o mecanismo de detecção de dependências utiliza as informações da *cache* simples para determinar se os seus operandos fonte são provenientes de instruções precedentes do mesmo bloco básico ou de um outro bloco.

Se o campo correspondente na *cache* de dependências estiver com "0", o operando fonte pode estar em um dos três casos: ele já pode ter sido gerado num ciclo anterior; ele ainda está sendo avaliado por uma instrução despachada num ciclo anterior; ou finalmente, ele será produzido por uma instrução precedente que encontra-se no mesmo *buffer* de despacho.

No algoritmo empregado na *cache* simples, em qualquer um desses três estados, para cada operando fonte da instrução armazenada na entrada índice  $k$  do *buffer*, o mecanismo de detecção examina os registradores destino das  $k - 1$  primeiras instruções do *buffer*.

Havendo uma instrução precedente cujo registrador destino será usado como fonte pela  $k$ -ésima instrução, o mecanismo detecta que essa é a instrução que produzirá o operando fonte, e o *tag* da estação de reserva escolhida para esta instrução é que será usado no lugar do operando fonte. Caso contrário, se o resultado já foi gerado em um ciclo anterior, o operando será lido do banco de registradores. Se ele ainda está sendo gerado por uma instrução despachada num ciclo anterior, o *tag* identificador da estação de reserva correspondente será fornecido no lugar do operando.

Ao especificarmos o mecanismo de detecção de dependências empregado na *cache* simples, estávamos interessados em reduzir o número de comparações sem ter que assumir os custos mais elevados exigidos pelos outros dois esquemas propostos que,

conforme será visto, são mais eficientes.

### 3.6.3 *Cache* de Dependências Inteligente

A estrutura básica é a mesma da *cache* simples, conforme mostrado na Figura 3.9, mudando apenas o algoritmo utilizado.

No modelo anterior, para cada dependência não resolvida, é necessário fazer comparações com todas as instruções precedentes que estão sendo despachadas no mesmo ciclo. Entretanto, as relações de dependências de cada instrução com as instruções precedentes que pertencem ao mesmo bloco básico já foram determinadas e armazenadas. Para os operandos com relações de dependências não definidas, isto é, com os grupos preenchidos com "0", só é necessário a comparação com instruções precedentes que pertençam a outros blocos básicos.

A primeira conclusão desta observação é que as instruções do primeiro bloco básico no *buffer* de despacho não necessitam de comparações, pois as instruções com dependências não resolvidas dependem de blocos básicos que já foram despachados em ciclos anteriores. Os seus operandos ou *tags* correspondentes serão buscados no banco de registradores.

Para os demais blocos básicos do *buffer* de despacho, o campo de índice que acompanha cada instrução é diminuído do seu índice no *buffer* de despacho. O resultado fornece o índice dentro do *buffer* da primeira instrução a partir da qual as comparações deverão ser feitas, ou seja, a última instrução do bloco básico anterior.

Estas melhorias no algoritmo reduzem ainda mais o número de comparações necessárias para despachar cada instrução, já que apenas são realizadas comparações com as instruções dos blocos precedentes, que estiverem no mesmo *buffer* de despacho.

Existe o custo de um *hardware* para calcular o endereço da primeira instrução a ser comparada, um somador de 4 *bits* para cada instrução do *buffer* de despacho. Além disto, um pequeno aumento na complexidade da lógica de controle é esperado, mas os custos de armazenamento são os mesmos que o da *cache* simples.

Quando a *cache* de dependências indica que existe uma relação de dependência com outra instrução do **mesmo** bloco básico, é preciso verificar-se ainda se esta instrução está no mesmo *buffer* de despacho ou se foi despachada em um ciclo

anterior. Para isto, diminui-se o índice extraído da *cache* de dependências do índice da primeira instrução do bloco básico que está armazenado no *buffer*. Se este valor for negativo, então a instrução que gera o respectivo operando foi despachada em um outro ciclo. Se o valor for positivo, ele indica a posição do *buffer* de despacho em que esta instrução está localizada.

Com esta informação, obtém-se o identificador da estação de reserva para onde esta instrução irá ser despachada e enviamos este valor no lugar do operando para estação de reserva onde a instrução dependente será armazenada. Este procedimento é válido para todos os modelos de *cache* de dependências.

### 3.6.4 Cache de Dependências Avançada

No modelo denominado *cache* de dependências avançada, outras informações são armazenadas em cada entrada da *cache*: quais os registradores que são alterados por cada bloco básico, e qual a última instrução que modificou cada registrador.

Tag de Endereço	Tamanho	Identificador de Dependências				
$32 - \log_2(N + 2) \text{ bits}$	4 bits	4 bits	4 bits	4 bits	6 bits	• • •
		Op. Fonte 1	Op. Fonte 2	Cód. de Cond.	Op. Destino	

Figura 3.10: Controle das Dependências Internas - Opção 1

Essa informação pode ser codificada com um sub-campo adicional de 6 *bits* no campo identificador de dependências (supondo-se 32 registradores inteiros e 32 de ponto flutuante), indicando qual o registrador alterado por cada instrução do bloco básico. Um esquema pode ser visto na Figura 3.10.

Este modelo de *cache* requer mais 96 *bits* para cada entrada da *cache* para armazenar esta informação adicional.

Outra forma de codificação seria utilizar uma máscara de 64 *bits* para cada entrada, indicando se o registrador foi alterado (neste caso colocando-se um *bit* '1') e em seguida um vetor com 4 *bits* por posição (uma para cada *bit* em '1' na máscara), indicando qual a última instrução que alterou o registrador. Em esquema pode ser visto na Figura 3.11

O custo de armazenamento deste esquema é variável, e depende do número de registradores que são alterados pelas instruções de cada bloco básico. De acordo

Tag de Endereço	Máscara	Índice Instr.		Índice Instr.
$32 - \log_2(N + 2) \text{ bits}$	64 bits	4 bits	• • •	4 bits

Figura 3.11: Controle das Dependências Internas - Opção 2

com o trabalho de Huang [77], cerca de 90% dos blocos básicos modificam menos do que cinco registradores distintos. Neste caso, o gasto adicional seria de 84 bits em cada entrada.

A *cache* avançada armazena as informações de dependência entre as instruções do bloco básico e ainda indica quais são as instruções que estão alterando valores no banco de registradores. Note que se duas instruções alteram o mesmo registrador arquitetural, apenas o índice da instrução mais recente no bloco básico necessita ser armazenado na cache de dependências.

Os identificadores de registradores armazenados são os arquiteturais, pois no estágio de despacho, em nossa máquina básica, os registradores ainda não foram renomeados.

Quando em um *buffer* de despacho houver mais de um bloco básico despachado (o que é comum em programas inteiros), esta informação será utilizado pelas instruções dos blocos sucessores para identificar as relações de dependência com as instruções dos blocos que estão armazenados antes no *buffer* de despacho.

A *cache* avançada é lida e as informações de dependência de todos os blocos básicos daquele *buffer* de despacho são recuperadas. O primeiro bloco, como vimos, pode ter todas as suas instruções despachadas, levando-se em conta apenas as dependências internas ao bloco básico. Os blocos sucessores extraem as dependências internas da *cache* de despacho e, para os campos preenchidos com "0", isto é, dependentes de resultados gerados por instruções em outros blocos básico, o seguinte procedimento é realizado:

- Verifica-se quais são os registradores fonte que cada instrução necessita e se estes são produzidos por algum dos blocos básicos antecessores, cuja informação foi extraída da *cache* de dependências;
- Em caso afirmativo, o índice da instrução que modifica o registrador é obtido através da *cache* "avançada". Se não, o operando é lido do banco de

registradores;

- São feitos cálculos similares à *cache* inteligente para verificar se a instrução com a qual existe dependência está no mesmo *buffer* de despacho;
- Se for o caso, o *tag* escolhido para a esta instrução é que será enviado para a estação de reserva, em caso contrário o operando ou *tag* armazenado no banco de registradores será lido;
- Se o bloco básico antecessor não estiver armazenado na *cache* de dependências, procede-se da mesma forma que no caso da *cache* inteligente, comparando-se com todas as instruções daquele bloco básico.

Com as informações adicionais é possível reduzir significativamente o número de comparações, além de permitir que apenas os resultados “vivos” na saída do bloco básico sejam enviados para o banco de registradores. Os demais valores serão capturados apenas pelas respectivas estações de reserva que estiverem aguardando por esses valores. Esta providência reduz a pressão sobre o banco de registradores.

### 3.7 Resumo

Neste capítulo apresentamos as modificações propostas ao algoritmo de Tomasulo para o despacho de múltiplas instruções por ciclo. Comparamos os nossos modelos com adaptações similares desenvolvidas para processadores comerciais.

O uso do despacho seletivo, como meio de reduzir o número de barramentos de resultados (CDB) e de comparadores nas estações de reserva, também foi apresentado. O Capítulo 5 apresenta dados sobre o número médio de dependências por instrução, que justificam os modelos apresentados.

Apresentamos também um exemplo que motivou a concepção e especificação de uma *cache* de dependências, mostrando o elevado número de comparações necessárias ao despacho de múltiplas instruções por ciclo. O objetivo de nosso modelo é reduzir o número destas comparações (e comparadores).

Os tamanhos dos campos limitam este esquema a um máximo de 16 instruções por bloco básico. Caso o bloco básico possua mais de 16 instruções, ele será subdividido em blocos com até 16 instruções cada. Esta limitação é bastante razoável se

considerarmos que para os programas de aplicações inteiras, o tamanho médio dos blocos básicos se situa em torno de 5 ou 6 instruções. No caso das aplicações onde o bloco básico é maior (e.g., ponto flutuante), essa limitação também não é relevante dado o número máximo de instruções que efetivamente podem ser executadas em paralelo com a tecnologia atual.

A decodificação da instrução permite identificar os grupos que contêm informações válidas. Uma instrução cuja decodificação indique que ela é independente, por conter apenas operandos imediatos, pode ser despachada sem uso das informações na *cache* de dependências.

# Capítulo 4

## O Ambiente Experimental

### 4.1 Introdução

Uma importante decisão para conduzir os experimentos desta tese foi a escolha do método de simulação a ser empregado. Algumas opções estavam disponíveis, e uma escolha equivocada poderia prejudicar ou inviabilizar o processo de modelagem de nossas máquinas experimentais.

Os aspectos “desempenho e confiabilidade” não poderiam ser ignorados durante a escolha do método: pretendíamos realizar (e realizamos) um grande número de simulações envolvendo a execução dos programas do conjunto SPEC95, cada simulação dedicada ao estudo do impacto no desempenho provocado pela inclusão de uma característica específica nas diversas configurações derivadas do nosso modelo básico.

Uma alternativa, seria desenvolver um simulador genérico, para reproduzir o comportamento da família de máquinas experimentais derivadas do nosso modelo de arquitetura. Nesse caso, o simulador processaria o código objeto do programa de teste e produziria os dados estatísticos desejados. Todavia, a complexidade dos processadores atuais impõe restrições a este método de simulação, pois a possibilidade de introdução de erros durante a programação do modelo é muito grande.

Na área de arquitetura de computadores, a tendência recente consiste em utilizar simulações baseadas em *trace*. Os *traces* são seqüências de endereços e instruções que reproduzem, total ou parcialmente, as instruções executadas pelos programas.

Esse método de simulação é confiável e permite reproduzir a execução dos progra-

mas nos sistemas investigados, pois os *traces* são gerados por simuladores confiáveis, normalmente de uma versão escalar da arquitetura que será investigada.

Usualmente, os *traces* produzidos são armazenados em arquivos. Alternativamente, eles podem ser produzidos e utilizados durante cada simulação realizada. O armazenamento em arquivos apresenta o inconveniente do grande espaço de armazenamento requerido. Por outro lado, a reprodução dos *traces* durante cada simulação pode ser demorada em alguns casos.

Depois de armazenados, os *traces* podem ser utilizados por outros simuladores que irão avaliar o efeito produzido pelas modificações introduzidas na arquitetura básica. Ao empregarmos o método de *traces*, estaremos simplificando a construção dos simuladores dos modelos propostos, porque os *traces* apresentam o fluxo correto de execução das instruções.

Há vários métodos para a geração de *traces*. Eles podem ser divididos em dois tipos: os baseados em *hardware* e os baseados em *software*, conforme pode ser visto na Tabela 4.1.

<b>Hardware</b>
Monitores e Instrumentação Microcódigo
<b>Software</b>
Baseados em <i>Trap</i> Emuladores Anotação de Código (fonte, objeto ou executável)

Tabela 4.1: Métodos para Coleta de *traces*

Antes de enumerar as vantagens e desvantagens de cada um destes métodos, vamos relacionar as métricas que devem ser consideradas durante a avaliação de cada método:

- Velocidade - Taxa de captura do *trace* ;
- Memória - Quantidade de memória requerida;
- Acurácia - Perturbação no endereçamento;
- Intrusividade - Sobrecarga do *trace*;



- Completeza - Inclusão de bibliotecas, interrupções e S.O. no *trace*;
- Granularidade - Menor unidade amostrável;
- Flexibilidade - Facilidade de uso;
- Portabilidade - Uso em diversas plataformas;
- Capacidade - Espaço gasto para armazenamento do *trace*.

## 4.2 Vantagens e Desvantagens dos Métodos

### 1. Monitores e Instrumentação

O uso de monitores e instrumentação por *hardware* tem a grande vantagem de capturar o *trace* na mesma taxa de execução do programa. Entretanto, para o armazenamento destes dados, é necessário o uso de *interleaving* e multiplexação para reduzir o efeito causado pela baixa taxa de transferência dos dispositivos de armazenamento de massa. As vantagens do método são a acurácia, não intrusividade e completeza (o *trace* inclui as instruções tanto da aplicação como do sistema operacional). As desvantagens são o elevado custo, tamanho limitado do *trace* que pode ser coletado e baixa portabilidade, já que cada plataforma exige um monitor específico.

### 2. Microcódigo

O método consiste em introduzir “ganchos” no microcódigo para capturar o estado da máquina. A vantagem do método é que ele permite capturar tanto o código do usuário como o do S.O. Ele também é rápido (apenas 2 a 10 vezes mais lentos que a execução normal). Como desvantagem podemos citar a necessidade de acesso ao microcódigo, a perda da portabilidade e o alto custo de implementação (demanda muita mão-de-obra).

### 3. Baseados em *Trap*

As ferramentas deste tipo obtêm o *trace* interrompendo a execução da aplicação em determinados pontos do programa para salvar as informações de *trace*. O método está disponível na maioria dos processadores, é portátil e de baixo

custo. Apresenta considerável lentidão na coleta dos *traces* (cerca de 1000 vezes mais lento), é intrusivo e incompleto.

#### 4. Emuladores

Simula as instruções da arquitetura alvo executando uma ou mais instruções de máquina na arquitetura hospedeira. Os aspectos favoráveis são: o pequeno impacto no tempo de execução do programa (de 10 a 100 vezes mais lentos); é portátil; flexível; e permite o *trace* no nível de instrução. Os fatores desfavoráveis são: a quantidade de memória adicional requerida; e apenas monitoram processos únicos pois não possuem suporte para *threads*. O emulador *Shade* [78], da Sun Microsystems, é um exemplo desse método. Ele traduz as instruções emuladas para as instruções nativas, e utiliza uma *cache* para armazenar e reutilizar as seqüências de instruções traduzidas.

#### 5. Anotação de Código

Programas instrumentados produzem o *trace* durante a execução da aplicação. Esta anotação pode ser realizada em três níveis: código fonte, código objeto e código executável. São relativamente fáceis de implementar, de baixo impacto na aplicação (10 vezes mais lento) e de baixo custo. A desvantagem do método é que ele não pode capturar as instruções do S.O. e das bibliotecas, e aumentam a quantidade de memória de cada programa.

Como exemplo de anotação de código objeto temos o ATOM+, da DEC, que executa em máquinas Alpha com sistemas operacionais tipo Unix e é capaz também de simular instruções do sistema operacional. Este simulador é independente da linguagem de programação ou do compilador; a informação de endereço não é perturbada; o *trace* é passado para o programa de análise por uma chamada de rotina; e a instrumentação pode ser feita no nível de rotina, bloco básico ou instrução.

### 4.3 O Método Escolhido

Os *traces* devem ser produzidos com cuidado e utilizados adequadamente. O usuário deve entender tanto do comportamento da aplicação quanto de sua influência sobre

os parâmetros em estudo. A leitura do código fonte é bastante útil no auxílio desta tarefa.

Alguns erros que ocorrem com frequência são: o uso de programas muito pequenos para, por exemplo, investigar o comportamento de uma *cache*; o uso de programas de ponto flutuante para avaliar preditores de desvio; tentar melhorar o desempenho de *loops* de espera de E/S, já que o programa espera o mesmo tempo, só que executa mais vezes o *loop*; tentar simular o multiprocessamento através do simples *flush* da *cache*.

A Tabela 4.2 apresenta um resumo das características dos diversos tipos de métodos de obtenção de *traces*.

	Atraso	Sist. Operacional	Tamanho de Amostra	Custo
Fonte	10X	não	> GB	Baixo
Objeto	10X	alguns	> GB	Baixo
Binário	10X	não	> GB	Baixo
Microcódigo	10X	sim	> GB	Médio
<i>Trap</i>	1000X	sim	Ilimitado	Médio
Emulação	10-100X	não	Ilimitado	Médio
Monitores	1X	sim	< GB	Alto

Tabela 4.2: Comparação dos Métodos para Coleta de *traces*

Um dos maiores problemas para a geração de *traces* é o tempo e o espaço de armazenamento gastos quando as aplicações são muito grandes, como ocorre com os programas do conjunto SPEC95.

Para contornar esta limitação, avaliamos em um trabalho anterior [79], o potencial da técnica de *trace* amostrado [80] para simulação dos programas do SPEC92. O uso desta técnica permite que apenas parte da execução do programa seja armazenada no *trace*, reduzindo-se assim as necessidades de armazenamento e o tempo de simulação. Com o uso de técnicas de amostragem, esta redução de espaço e tempo se faz sem perdas significativas nas características das instruções em termos da frequência de execução ou mesmo do paralelismo no nível de instrução.

Contudo, preferimos adotar o simulador *Shade* [78, 81], desenvolvido pela Sun Microsystems para emular as arquiteturas SPARC V8 e V9. A grande vantagem desse emulador é a velocidade de simulação, que dispensa o armazenamento dos *traces*, já que estes podem ser obtidos com a realização de uma nova simulação.

Além disto, ele está disponível gratuitamente e executa nas plataformas que usamos (Sun UltraSPARC).

Com o uso do *Shade*, os usuários podem escrever programas, chamados *analisadores*, que invocam funções do *Shade* para simular a arquitetura SPARC e obter *traces* das aplicações, usualmente *benchmarks*. Um uso comum para os analisadores é simular *caches* e *pipelines* de microprocessadores.

A velocidade de emulação do *Shade* é cerca de 2 a 6 vezes mais lentas que a execução normal. A coleta de *trace* e os analisadores diminuem este desempenho, mas a quantidade de informação a ser coletada é determinada pelo usuário.

Com o *Shade* podemos obter vários tipos de informação: um *trace* dinâmico com todas instruções ou apenas, por exemplo, das instruções de desvio; um arquivo de saída contendo o tamanho e quantas vezes cada bloco básico foi executado; ou ainda a frequência de execução das instruções e dos blocos básicos de um programa.

É possível o acesso, após a execução de cada instrução, ao estado do programa simulado (registradores e memória). Várias informações podem ser obtidas através do *Shade*: qual o tipo de instrução executada; se foi um *trap* ou desvio condicional; se este foi tomado ou não; qual o endereço da instrução alvo; e qual o PC atual.

Ao invés de *pipes* ou arquivos, as informações coletadas (i.e. os *traces*) são transmitidas do *Shade* para os analisadores através da memória. O *Shade* gera essas informações com rapidez e o nível de detalhamento desejado é programável. Desse modo, os *traces* podem ser recriados por demanda, evitando assim os custos de processamento e armazenamento dos *traces* convencionais.

É importante ressaltar que as chamadas ao sistema são executadas corretamente, isto é, os serviços solicitados ao S.O são atendidos corretamente. O programa de aplicação pode, por exemplo, ler e escrever em arquivos, alocar e liberar memória. Contudo, estes serviços são executados pelo sistema operacional da máquina hospedeira, e por esse motivo as instruções que compõem as chamadas do S.O. **não** são contabilizadas pelo *Shade*.

Segundo Cvetanovic [82], o percentual do tempo de execução das funções do Sistema Operacional é menor que 3% para a maioria dos programas inteiros e de ponto flutuante do SPEC92. Logo, essa restrição, que foi observada também em outros trabalhos, não deve alterar significativamente os dados obtidos experimentalmente.

## 4.4 O Ambiente de Coleta de Dados

O *Shade* foi utilizado durante duas fases de nossa pesquisa de tese. A primeira fase determinou as características dos programas inteiros e de ponto flutuante do SPEC95. Esta fase foi importante pois resultou no levantamento de informações que determinaram a direção de nossa pesquisa. A segunda fase avaliou os ganhos obtidos com a introdução de modificações propostas no nosso modelo básico de arquitetura super escalar.

Para a primeira fase desenvolvemos um analisador, com cerca de quinhentas linhas de código *C*, com o objetivo de determinar algumas características dos programas do SPEC95: o tamanho dos blocos básicos, distância média entre desvios tomados, entre outros. Nesta fase, os programas são executados até a ocorrência de uma instrução de desvio, *trap* ou de uma chamada de subrotina. Quando isto ocorre, a execução é interrompida, o controle é passado para o analisador, e os dados relativos àquele bloco básico são acrescentados às estatísticas.

Para a segunda fase desenvolvemos um simulador/analisador para o modelo proposto. Este simulador, com cerca de 1100 linhas de código *C* e avalia a eficiência da modificação introduzida no modelo de máquina. Nesta simulação, a execução dos programas é interrompida a cada instrução. Os dados da instrução simulada são transferidos para o simulador, que modifica o estado da máquina em estudo, e coleta os dados estatísticos apropriados.

O *Shade* pode ser visto como uma grande biblioteca de rotinas, que tem como função a emulação do código objeto, mediante determinados parâmetros de controle, passados pelo usuário através da programação de um analisador. Assim, o código do analisador é compilado, e o código relativo ao *Shade* integrado na forma de chamadas a rotinas de biblioteca.

O diagrama da Figura 4.1 ilustra como os analisadores e o *Shade* são integrados, como os programas objeto são simulados, e como as estatísticas são coletadas.

Equipamos os simuladores das máquinas experimentais com recursos ilimitados: número de unidades funcionais; capacidade da memória *cache*; capacidade das *caches* de dependências; etc. Os dados produzidos por nossas máquinas experimentais representam o desempenho que poderá ser obtido em condições ideais. Na prática, os resultados devem ser mais modestos.

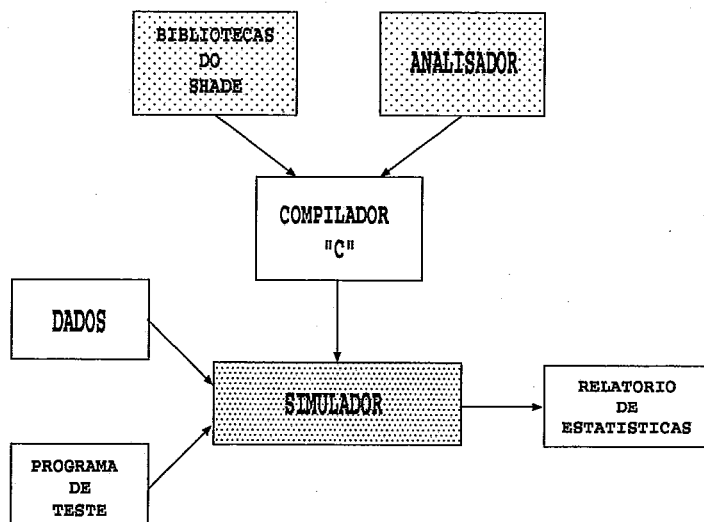


Figura 4.1: Método de Simulação

Para avaliar as máquinas derivadas do modelo proposto, foram utilizados os programas inteiros e de ponto flutuante do SPEC95. Os programas foram compilados para a arquitetura SPARC, sem fazer o uso da “janela” de registradores e do *delay slot*.

Os programas inteiros e de ponto flutuante do conjunto SPEC95, foram simulados integralmente, ora com a entrada *reference*, ora com a entrada *training*. A entrada que utilizada é sempre indicada antes da apresentação dos resultados dos experimentos.

À medida que aumentava a complexidade das máquinas experimentais, foi necessário empregar uma entrada menor (*training*) para algumas simulações. Desse modo, o tempo de simulação tornou-se viável. Essa decisão não é prejudicial para nossa modelagem, pois conforme mostrado em [83], o percentual de instruções comuns que são executadas com os dois tipos de entrada é alto, entre 88% e 100% para a maior parte dos programas do SPEC95. A exceção fica por conta dos programas *perl* (63,8%) e *apsi* (74,3%).

Durante nossos experimentos, apenas as instruções das rotinas do Sistema Operacional não foram simuladas, por limitação do emulador *Shade*. Apesar da velocidade do simulador, foram requeridas centenas de horas de simulação em várias estações de trabalho do tipo Ultra-1, para a realizar os experimentos descritos nesta tese.

## 4.5 Características dos Programas de Avaliação

### 4.5.1 Introdução

Empregar uso de uma bateria de programas de teste (*benchmarks*) durante o desenvolvimento de novas arquiteturas é um procedimento usual. Entretanto, para uma correta interpretação dos resultados obtidos é necessário conhecer detalhadamente as características dos programas de teste. No desenvolvimento desta tese, utilizamos os programas inteiros e de ponto flutuante do conjunto SPEC95 [84].

Os programas do SPEC95 se destinam primariamente à avaliação do desempenho de processadores, exercitando as unidades funcionais inteiras e de ponto flutuante. São programas escolhidos dentre as aplicações reais e têm como objetivo testar os principais componentes da arquitetura de um processador. Nos experimentos descritos nesta tese, os programas foram traduzidos pelos compiladores C (programas inteiros) e FORTRAN (programas de ponto flutuante) da SUN, versão 3.0, e o código objeto gerado é compatível com a arquitetura SPARC, versão V8.

Nas medidas realizadas, quando mencionarmos o termo “código estático” estamos considerando o conjunto de instruções de máquina que formam o programa após a compilação. Dependendo do fluxo de controle do programa, algumas destas instruções nunca serão executadas, outras serão executadas diversas vezes. Denominamos “código dinâmico”, o conjunto de instruções que forma o fluxo de execução do programa, ou seja, o total de instruções executadas pelo programa.

Durante a avaliação das características dos programas de teste, empregamos o método de *profiling* para determinar as rotinas mais executadas de cada programa. Esse método introduz código no início e no término de cada rotina, para o registro do perfil de execução do programa. Para os programas de ponto flutuante, o tempo gasto na execução das subrotinas é apresentado sob a forma de percentagem do tempo total de execução do programa. No caso dos programas inteiros, citamos apenas as rotinas mais executadas, porque o método de *profiling* é de grande peso no tempo total de execução do programa e não permitiu a obtenção de valores percentuais acurados.

Para obter o restante das características, os programas foram simulados com a entrada *reference* ou *training* do SPEC95 no ambiente *Shade* [78] até o seu término.

As exceções foram os programas inteiros *gcc* e *vortex* durante a primeira fase de nosso estudo. O *gcc* foi simulado usando como entrada apenas um dos programas do conjunto de entrada de dados e o último teve a entrada alterada para uma versão mais reduzida. Para a segunda fase dos estudos, as entradas de dados foram utilizadas sem modificação.

## 4.5.2 Programas de Ponto Flutuante

O SPEC95 é composto por dez programas de ponto flutuante: *tomcatv*, *swim*, *su2cor*, *hydro2d*, *mgrid*, *applu*, *turb3d*, *apsi*, *fpppp* e *wave*. Neste seção as seguintes características desses programas foram determinadas:

- Total de instruções do código estático;
- Total de instruções do código dinâmico;
- Tamanho médio dos blocos básicos do código estático;
- Tamanho médio dos blocos básicos do código dinâmico;
- Rotinas mais executadas.

O tamanho médio dos blocos básicos do código estático é a média aritmética do tamanho de todos blocos básicos do código objeto. Sob o ponto de vista dinâmico, esse tamanho é a média ponderada do tamanho dos blocos básicos que foram executados, considerando-se como peso o número de vezes que cada bloco básico foi executado. Por fim, determinamos quais são as rotinas de maior peso no tempo total de execução de cada programa. Este último dado é proveniente das informações obtidas com o método *profilng*. As Tabelas 4.3 e 4.4 reúnem as características dos programas de ponto flutuante.

### Tomcatv

O *tomcatv* é um programa em precisão dupla de geração de malha e é parte da bateria de avaliação do Prof. W. Gentsch. No SPEC95 o programa teve o seu gerador de entradas modificado e agora gera uma malha bem maior. É um dos *benchmarks* mais sensíveis ao tempo de acesso à memória do sistema e pode ser



Aplicação	Código Estático	Código Dinâmico
tomcatv	48.000	23 bilhões
swim	56.000	12 bilhões
su2cor	77.000	42 bilhões
hydro2d	62.000	73 bilhões
mgrid	51.000	22 bilhões
applu	73.000	67 bilhões
turb3d	63.000	94 bilhões
apsi	100.000	50 bilhões
fp PPP	77.000	220 bilhões
wave	115.000	45 bilhões

Tabela 4.3: Total de Instruções - Ponto Flutuante

Aplicação	Código Estático	Código Dinâmico
tomcatv	3,0	17,7
swim	3,2	38,0
su2cor	3,9	20,3
hydro2d	3,5	11,6
mgrid	3,1	11,6
applu	4,3	18,3
turb3d	3,5	7,5
apsi	4,6	24,4
fp PPP	4,3	43,9
wave	5,4	20,8
média	4,1	25,6

Tabela 4.4: Número Médio de Instruções por Bloco Básico - Ponto Flutuante

paralelizado razoavelmente bem. O código fonte em FORTRAN é formado por apenas 150 linhas, mas o código objeto tem 48.000 instruções (estático) e o dinâmico 23 bilhões. O tamanho médio do bloco básico, para os casos estático e dinâmico, é de 3,0 e 17,7 instruções, respectivamente. O programa *tomcatv* não tem rotinas e passa a maior parte do tempo executando a rotina principal (95%). Os 5% restantes são gastos em rotinas de bibliotecas e do Sistema Operacional.

## Swim

É um programa de predição de clima. Ele resolve um sistema de equações de águas rasas utilizando o método das diferenças finitas. No SPEC95, a matriz foi expandida para 1024x1024 e diversas instabilidades numéricas foram reparadas. Cada bloco básico possui 3,2 instruções em média (código estático) ou 38,0 instruções (código

dinâmico). O código estático é formado por 56.000 instruções e o código dinâmico por 12 bilhões de instruções executadas. É um programa em precisão simples e com 40% do tempo de execução gasto na rotina principal.

### **Su2cor**

O *su2cor* é um programa em precisão dupla que calcula as massas de partículas elementares segundo a teoria de Quark-Gluon. Ele usa o método de Monte-Carlo e pode ser razoavelmente paralelizado. Verificou-se uma média estática de 3,9 instruções por bloco básico e uma média dinâmica de 20,3 instruções. Ele possui 77.000 instruções no código estático e 42 bilhões de instruções no código dinâmico.

### **Hydro2d**

É um programa de astro-física em precisão dupla que usa as equações hidrodinâmicas de Navier Stokes para calcular jatos galáticos. Pode ser razoavelmente paralelizado e a rotina mais executada é a FILTER (47%), que limita os fluxos anti-difusão. O tamanho médio dos blocos básicos é de 3,5 e 11,6 instruções, respectivamente, para os códigos estático e dinâmico. O programa possui 62.000 instruções no código estático e um total de 73 bilhões executadas.

### **Mgrid**

Programa para solucionar uma *multi-grid* em um campo potencial em 3D. É um dos programas de paralelização mais fácil do SPEC95. A rotina mais executada é a RESID (55%), que calcula o valor residual ( $R = V - AU$ ), onde  $R$ ,  $V$  e  $U$  são matrizes tri-dimensionais e  $A$  um vetor de quatro componentes, todos em precisão dupla. O total de instruções dinâmicas do programa é de 22 bilhões, e o estático 51.000 instruções. O tamanho dos blocos básicos é de 3,1 instruções em média (código estático) ou de 11,6 instruções em média (código dinâmico).

### **Applu**

Programa em precisão dupla que resolve equações diferenciais parciais parabólicas-elípticas. O programa executa 67 bilhões de instruções e o código estático inclui 73.000 instruções distintas. O tamanho médio dos blocos básicos é de 4,3 instruções

(código estático), ou de 18,3 instruções, se considerarmos o fluxo de execução das instruções. A rotina principal consome 65% do tempo de execução.

### **Turb3d**

Programa em precisão dupla que simula uma turbulência homogênea e isotrópica em um cubo com condições de contorno periódicas na direção das coordenadas x, y e z. Ele resolve as equações de Navier-Stokes usando um método pseudo-espectral. O código é praticamente todo paralelizável. A rotinas mais executadas é FFTZ2 (24%), que realiza a  $L$ -ésima iteração da segunda variação da FFT de Stockham. A rotina DCFT consome 18% do tempo de execução e realiza uma transformada de Fourier em 3-D com dados complexos na direção X,Y. O tamanho médio dos blocos básicos é de 3,5 instruções (código estático) e de 7,5 instruções (código dinâmico). O programa objeto possui cerca de 63.000 instruções (código estático) e requer a execução de 94 bilhões de instruções.

### **Apsi**

Programa em precisão dupla que resolve problemas relativos a temperatura, vento, velocidade e distribuição de poluentes. Tem origem na bateria de teste *Perfect Club* e não é paralelizado facilmente. O total estático de instruções da aplicação é de 100.000 e o total dinâmico é de 50 bilhões de instruções. O tamanho médio dos blocos básicos é de 4,6 instruções (código estático) de 24,4 instruções (código dinâmico).

### **Fpppp**

Ele é proveniente da série Gaussiana de programas de avaliação de química quântica. Os átomos no *fpppp* são colocados em uma região relativamente compacta do espaço, posicionadas em um *lattice* do tipo grafite. Não é paralelizado em alto nível, mas exibe boas possibilidades de exploração de paralelismo em ambientes super escalares. O tamanho médio estático dos blocos básicos é de 4,3 instruções e o dinâmico de 43,9 instruções. O código estático possui 77.000 instruções que executadas totalizam 220 bilhões de instruções. A rotina mais executada é a TWLDRV (53%), que calcula a contribuição integral de dois elétrons para o conjunto das forças, em precisão dupla.

## Wave

O programa de física plasmática *wave* soluciona equações de Maxwell e equações de movimento de partículas eletromagnéticas em uma malha cartesiana com uma variedade de campos e partículas como condições de fronteira. A rotina mais executada, PARMVR (60%) é um deslocador relativístico de partículas, em precisão dupla. O tamanho médio estático dos blocos básicos é de 5,4 instruções e o dinâmico de 20,8 instruções. O código estático possui cerca de 115.000 instruções e o dinâmico em torno de 45 bilhões de instruções.

### 4.5.3 Programas Inteiros

O SPEC95 é composto por oito programas inteiros: *go*, *gcc*, *li*, *perl*, *m88ksim*, *compress*, *jpeg* e *vortex*. A seguir avaliamos as seguintes características destes programas:

- Total de instruções do código estático;
- Total de instruções do código dinâmico;
- Tamanho médio dos blocos básicos (código estático);
- Tamanho médio dos blocos básicos (código dinâmico);
- Rotinas mais executadas.

Como anteriormente, o tamanho médio dos blocos básicos (estático) é a média aritmética do tamanho de todos blocos básicos do programa. O tamanho médio dinâmico dos blocos básicos é a média ponderada do tamanho dos blocos básicos que foram executados, considerando-se como peso o número de vezes que cada bloco básico foi executado. As Tabelas 4.5 e 4.6 apresentam as características dos programas inteiros do conjunto SPEC95.

O tamanho médio dos blocos básicos aumenta bastante se considerarmos o número de vezes que cada bloco básico foi executado. A média dinâmica é de 4,9 e a média estática é de apenas 3,3. Estes valores são importantes para determinar o tamanho da linha de uma *cache* de blocos básicos [74].

Aplicação	Código Estático	Código Dinâmico
go	88.000	37 bilhões
gcc	340.000	1,7 bilhão
m88ksim	56.000	25 bilhões
compress	31.000	45 bilhões
li	43.000	59 bilhões
jpeg	67.000	42 bilhões
perl	102.000	18 bilhões
vortex	165.000	0,9 bilhão

Tabela 4.5: Total de Instruções - Prog. Inteiros

Aplicação	Código Estático	Código Dinâmico
go	3,8	5,4
gcc	3,6	4,2
m88ksim	3,2	3,7
compress	3,0	4,3
li	3,2	3,8
jpeg	3,3	7,6
perl	3,5	4,2
vortex	3,2	4,8
média	3,3	4,9

Tabela 4.6: Numero Médio de Instruções por Bloco Básico - Prog. Inteiros

## Go

O programa *go* envolve as áreas de inteligência artificial, comparação de padrões, predição, etc. Ele é baseado no programa *The Many Faces of Go* de David Fotland. Como *benchmark* é justamente um teste para avaliar o desempenho da unidade de aritmética inteira. O uso de *inlining* das rotinas de manipulação de listas é recomendado, pois a maior parte do tempo de execução é gasto nestas rotinas. Caso contrário o tempo de processamento pode aumentar em até 30%. A execução do programa totaliza 37 bilhões de instruções e o código estático é formado por 88.000 instruções. O tamanho médio dos blocos básicos é de 3,8 e 5,4 instruções, para os códigos estático e dinâmico, respectivamente.

## m88ksim

Simula o processador MC88100 da Motorola. É uma implementação razoável de um simulador completo de processador, incluindo latências de *cache*. O simulador

é carregado com versões dos programas *Drhystone* e de um teste de memória. O total de instruções executadas é de 25 bilhões, para um código estático com 56.000 instruções. O tamanho médio dos blocos básicos é de 3,2 e de 3,7 instruções, para os código estático e dinâmico, respectivamente. As rotinas mais executadas são *killtime*, que é chamada sempre que uma instrução simulada precisa esperar por um operando ficar pronto, e *data\_path*, que é o simulador do caminho de dados do MC88000, e é o coração do simulador, reproduzindo a execução real da instrução para todas as instruções que não modificam o ponteiro de instruções (PC).

## Gcc

Baseado no compilador GNU C versão 2.5.3 da Free Software Foundation, este programa gera código para a arquitetura SPARC e usa cerca de 50 arquivos de entrada. Isto implica que ele é o *benchmark* com o maior número de *fork/exec* durante sua execução, e também com maior número de operações do tipo *open* e de outras chamadas ao Sistema Operacional. O *gcc*, juntamente com o *vortex*, é um dos maiores programas de teste do conjunto SPEC95. Para o estudo aqui realizado ele foi executado com uma entrada reduzida, que resultou em apenas 1,7 bilhões de instruções executadas, medindo-se um tamanho médio de bloco básico de 4,2 instruções, contra 3,6 instruções no código estático. O código estático possui 340.000 instruções. A rotina mais executada é a de eliminação de expressões comuns, mas que não possui peso muito grande no tempo total de execução do programa.

## Compress

Baseado no utilitário Unix de compressão de arquivos, esta versão foi modificada para fazer o processamento em memória, ao invés de ler/escrever arquivos. O programa gera um grande *buffer* de dados, comprime estes dados em outro *buffer* em memória, verifica o resultado e descomprime de volta para o *buffer* de memória. A partir de um código estático com 31.000 instruções, são executadas um total de 45 bilhões de instruções. Ele apresentou uma média de 3,0 instruções por bloco básico no código estático e de 4,3 instruções no código dinâmico. A maior parte do tempo de execução concentra-se nas rotinas de compressão/descompressão.

## Li

Basicamente é o mesmo interpretador Xlisp do SPEC92. Contudo, no SPEC95, a carga de trabalho é muito mais significativa: ele processa uma versão dos programas de avaliação de Richard Gabriel, extraídos de *Performance Evaluation of Lisp Systems*. A execução de diversas rotinas de pequeno tamanho é freqüente ao longo do programa. Em termos do tempo de execução a rotina mais cara é *mark*. O código estático possui 43.000 instruções e o total de instruções executadas é de 59 bilhões. Cada bloco básico possui em média 3,2 instruções (código estático) e 3,8 instruções (código dinâmico).

## Ijpeg

Programa de compressão/descompressão de imagens em memória, baseado nas especificações JPEG. Da mesma forma como no programa *compress*, ele foi adaptado para manipular as informações da memória ao invés de ler/escrever arquivos. O programa realiza uma série de compressões, com diferentes níveis de qualidade sobre uma variedade de imagens. A carga de trabalho foi dimensionada tomando-se como parâmetro um melhor compromisso entre tempo e espaço para uma variedade de imagens. O programa pode apresentar um bom desempenho em arquiteturas super escalares inteiras. A rotina mais executada é a de multiplicação de inteiros da biblioteca, já que a arquitetura SPARC possui apenas a instrução *MULScC* (*multiply step and modify condition codes*), sem possuir unidade de multiplicação de inteiros. O tamanho médio dos blocos básicos é de 3,3 instruções no código estático e 7,6 instruções no código dinâmico. São executadas 42 bilhões de instruções, a partir de um código estático com 67.000 instruções.

## Perl

É um interpretador para a linguagem PERL. Esta versão teve muito de suas características do sistema operacional Unix removidas para simplificar a execução em outros sistemas operacionais. São executados *scripts* para determinar por exaustão os números primos em um determinado conjunto de dados e buscar quais as palavras válidas em um dicionário a partir de um conjunto de letras. A rotina mais cara em termos de tempo de execução é *eval*, para avaliar expressões, que possui uma quan-

tidade razoável de instruções de ponto flutuante. A maior parte do tempo restante é gasto em rotinas usualmente encontradas na biblioteca *libc*: *malloc*, *free*, *memcpy*, etc. O tamanho médio dos blocos básicos é de 3,5 e 4,6 instruções, respectivamente, para os códigos estático e dinâmico. O total de instruções do código estático é de 102.000 e são executadas um total de 18 bilhões de instruções.

## Vortex

O programa de avaliação *vortex* originou-se de um programa de manipulação de banco de dados orientados a objeto chamado VORTEX. O *benchmark* constrói bases de dados separadas, mas interrelacionadas, construídas a partir de definições em esquemas texto-arquivo. A carga de trabalho do VORTEX foi modelada para aumentar as variações no *mix* de transações. Este programa de avaliação, juntamente com o *gcc*, é um dos maiores do conjunto SPEC95. Nos testes realizados, empregamos uma entrada reduzida, executando um total de 872 milhões de instruções, a partir de um código estático com 165.000 instruções. O tamanho médio dos blocos básico do código estático é de 3,3 instruções. Este tamanho médio aumenta para 4,8 instruções no código dinâmico.

## 4.6 Resumo

Neste capítulo apresentamos diversos métodos de simulação. Para selecionar o simulador a ser utilizado, realizamos um levantamento das características dos métodos de simulação disponíveis.

Para lidar adequadamente com a grande quantidade de dados gerados, métodos de obtenção de *trace* por amostragem foram estudados e relatados em [79]. Contudo, a identificação de emuladores mais rápidos e que dispensavam o armazenamento de *traces* nos levou a optar pelo emulador *Shade* da arquitetura SPARC [78].

A flexibilidade e desempenho obtidos superaram nossas expectativas e nos possibilitaram obter os dados necessários a realização desta tese.

Os programas do conjunto SPEC95 são amplamente difundidos e utilizados na modelagem de novas arquiteturas. Nesta seção caracterizamos estes programas, de modo a tornar claro os aspectos de uma arquitetura que podem ser avaliados por



cada um destes programas.

A literatura corrente indica que os desvios condicionais são um dos grandes obstáculos à extração do paralelismo no nível de instrução. Entretanto, para os programas em teste, nossos experimentos mostraram que o tamanho médio “dinâmico” dos blocos básicos é maior que o estático, mesmo para os programas inteiros. Ou seja, os blocos básicos com menor tamanho tendem a não ser utilizados durante a execução do programa.

Para os programas de ponto flutuante, as características obtidas nos experimentos mostraram que o tamanho do bloco básico dinâmico é muito maior que o estático (como ocorreu com os programas inteiros), variando entre 7,5 e 44 instruções, com média ponderada de 25,6 instruções. Este tamanho de bloco básico é suficiente para a exploração de paralelismo para as largura de despacho abordadas em nossa tese.

# Capítulo 5

## Resultados dos Experimentos

### 5.1 Introdução

Alguns experimentos preliminares foram apresentados no Capítulo 4. Eles serviram para caracterizar os programas de teste: para cada programa determinamos o tamanho do código objeto, o total de instruções executadas e o tamanho médio dos blocos básicos do programa.

Neste capítulo descreveremos outro conjunto de experimentos, desta vez envolvendo nosso modelo básico. Realizamos simulações para avaliar:

- a distância média entre dois desvios tomados;
- quantos blocos básicos são responsáveis por 90% das instruções executadas;
- a localidade (espacial e temporal) apresentada pelos programas de teste.

Através desses experimentos foi possível quantificar tanto a dimensão das estruturas de armazenamento como o volume do paralelismo de baixo nível apresentado pelos programas.

Além desses conjuntos de experimentos, investigamos a viabilidade do despacho seletivo no nosso modelo. Finalmente, realizamos um outro conjunto de experimentos para avaliar o impacto provocado pelos nossos mecanismos de detecção de dependências no desempenho do modelo básico.

Nestes dois últimos conjuntos de experimentos utilizamos todos programas do SPEC95 e, para tornar viáveis os tempos de simulação, empregamos a entrada *training* do SPEC95, nos demais experimentos, a entrada *reference* foi utilizada.

## 5.2 Exploração de Paralelismo de Baixo Nível

### 5.2.1 Introdução

No capítulo anterior apresentamos dados sobre o tamanho médio dos blocos básicos das aplicações inteiras do SPEC95 e observamos que esses tamanhos são bem menores do que os valores apresentados pelos programas de ponto flutuante.

Os blocos básicos das aplicações de ponto flutuante dispensam esquemas sofisticados de predição de desvio e/ou busca de instruções, pois eles já apresentam um número suficiente de instruções para a exploração do paralelismo de baixo nível.

Blocos básicos com pequeno número de instruções exigem esquemas de predição de desvio mais sofisticados. Além desta desvantagem, o grande número de desvios torna a implementação dos dispositivos de armazenamento para estas predições mais complexa. Entretanto, teriam realmente os programas do SPECint95 características tão adversas à extração do paralelismo de baixo nível?

Para responder a esta questão, realizamos simulações investigando: distância média entre desvios tomados; número de blocos básicos responsáveis por 90% das instruções que o programa executou; e a localidade espacial e temporal dos programas.

Se o número de blocos básicos responsáveis por 90% das instruções executadas for pequeno, o custo dos dispositivos para predição de desvio poderia cair consideravelmente.

### 5.2.2 Distância Média entre Desvios Tomados

Com o objetivo de avaliar essa distância, nosso simulador foi instrumentado para monitorar a execução das instruções de desvios condicionais durante o processamento de cada programa do conjunto SPEC95. Duas variáveis foram usadas na realização das simulações:

- contador de instruções entre desvios;
- vetor distância.

A primeira variável armazena o número de instruções que foram executadas a partir do último desvio tomado. Quando da simulação de uma nova instrução o

contador é incrementado e o simulador verifica o tipo da instrução.

Se ela for um desvio, o simulador aguarda pelo seu resultado e se ele foi tomado, o contador já contém o total de instruções executadas entre os dois últimos desvios tomados.

A  $i$ -ésima entrada do vetor distância armazena o número de vezes que seqüências formadas por  $i$  instruções (i.e., entre dois desvios tomados) foram executadas. Desse modo, toda vez que um desvio é tomado, o conteúdo da variável contador de instruções é usado para indexar o vetor distância. Em seguida, essa entrada do vetor é incrementada, o contador de instruções entre desvios é zerado e uma nova iteração do processo de avaliação é iniciada.

Aplicação	Média	Moda
go	13,0	5
gcc	7,5	4
m88ksim	10,1	14
compress	9,2	3
li	6,5	4
ijpeg	14,3	10
perl	9,8	10
vortex	8,7	4
média	10,2	—

Tabela 5.1: Distância Média Entre Desvios Tomados

No final da simulação do programa, o vetor distância é usado no cálculo da média ponderada das distâncias entre dois desvios tomados e da distância mais freqüente (moda). A Tabela 5.1 apresenta os valores obtidos com a simulação dos programas inteiros.

A segunda coluna da Tabela 5.1 apresenta a média do número de instruções entre dois desvios tomados. A terceira coluna exibe a moda, ou seja, o valor mais freqüente.

Examinando esta tabela, podemos verificar que o número de instruções entre dois desvios tomados é maior que o número médio de instruções dos blocos básicos do código dinâmico (vide Tabela 4.6 na Seção 4.5.3). A última linha da Tabela 5.1 apresenta a média ponderada das distâncias dos programas inteiros. Essa média é de 10,2 instruções entre dois desvios tomados, ou seja, duas vezes maior que o tamanho médio dos blocos básicos para o código dinâmico. Isto indica que, em média, para

cada desvio tomado existe um que não foi tomado.

Os valores de moda que ocorrem com mais frequência são 4 e 10. Ou seja, ou a maior parte dos desvios é logo tomado (moda 3, 4 e 5), ou então o desvio no final do próximo bloco básico é que será tomado (moda 10 e 14).

A seguir, mostramos os gráficos com a distribuição das distâncias entre desvios tomados por cada um dos programas. No eixo horizontal estão representados os valores das distâncias entre 0 e 256 instruções. A escala é logarítmica e distâncias maiores que 256 foram contabilizadas como 256 (como no caso do programa *jpeg*). O eixo vertical apresenta o total de ocorrências de cada distância, representado em uma escala linear. Os programas inteiros foram executados com a entrada *referência* do SPEC95, exceto os programas *gcc* e *vortex* que utilizaram uma entrada reduzida, como descrito anteriormente.

Os gráficos exibidos apresentam um traçado bem irregular. Ao contrário do esperado, eles não apresentam um valor máximo, decrescendo suavemente até o eixo horizontal. Na realidade, eles podem apresentar duas ou mais distâncias que ocorreram um grande número de vezes.

Os programas *gcc/li* (Figura 5.1) são os que mais se aproximam do comportamento esperado, com distâncias máximas em torno de 4 instruções entre desvios tomados, decrescendo até que não apareçam mais valores significativos acima de 20 instruções.

Apesar da moda igual a 3, o programa *compress* (Figura 5.2) possui distâncias com 7, 9 e 11 instruções ocorrendo um grande número de vezes, apresentando ainda valores observáveis para distâncias com até 60 instruções entre desvios tomados.

Os resultados para o programa *vortex*, mostrados na Figura 5.2, indicam que a moda para a distância entre desvios tomados é de 4 instruções. A distância entre desvios tomados com 7 instruções ocorre também um número significativo de vezes.

O programa *go* (Figura 5.3) apresenta moda igual a 5 instruções entre desvios tomados, mas possui distâncias com 11 e 22 instruções que ocorrem um número elevado de vezes durante a execução do programa.

A moda do programa *perl* (Figura 5.3) é igual a 10 instruções entre desvios, com distâncias iguais a 3, 7 e 9 instruções muito frequentes, decrescendo irregularmente até que, para distâncias acima de 40, o número de ocorrências é pouco significativo.

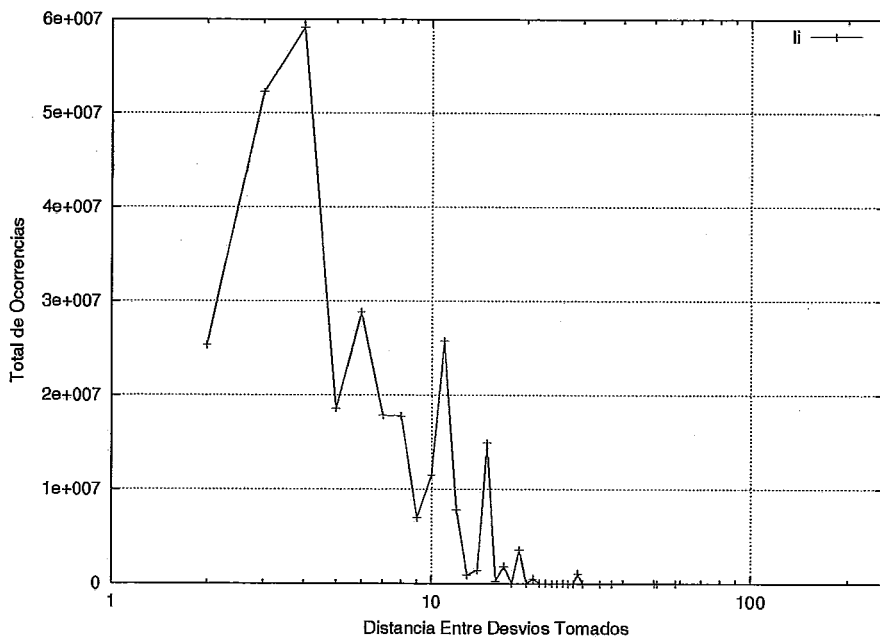
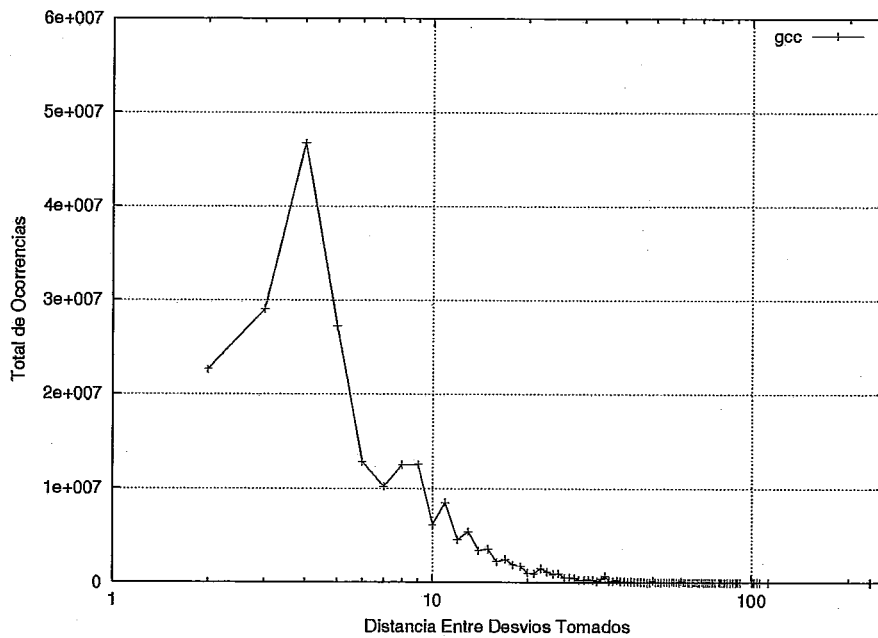


Figura 5.1: Distância Entre Desvios Tomados - gcc e li

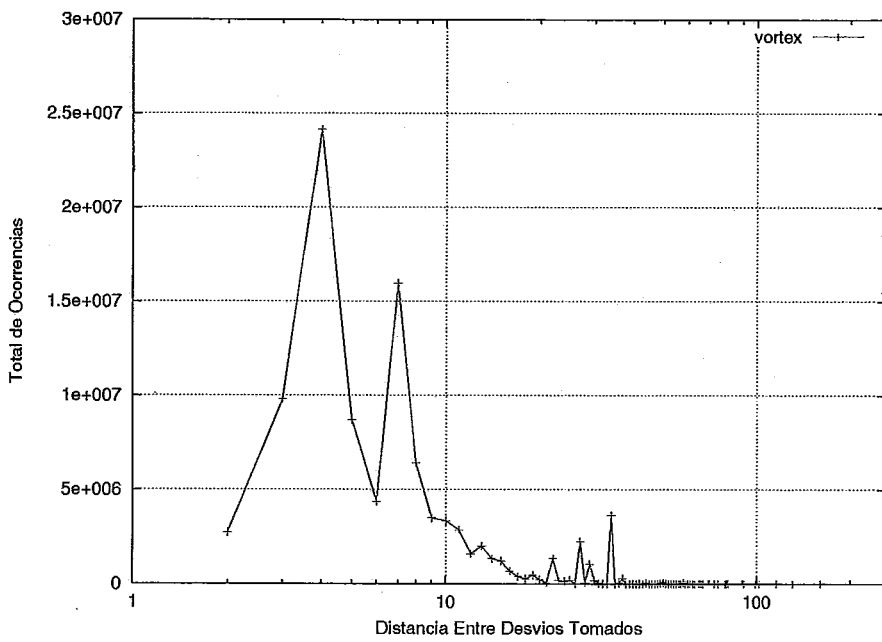
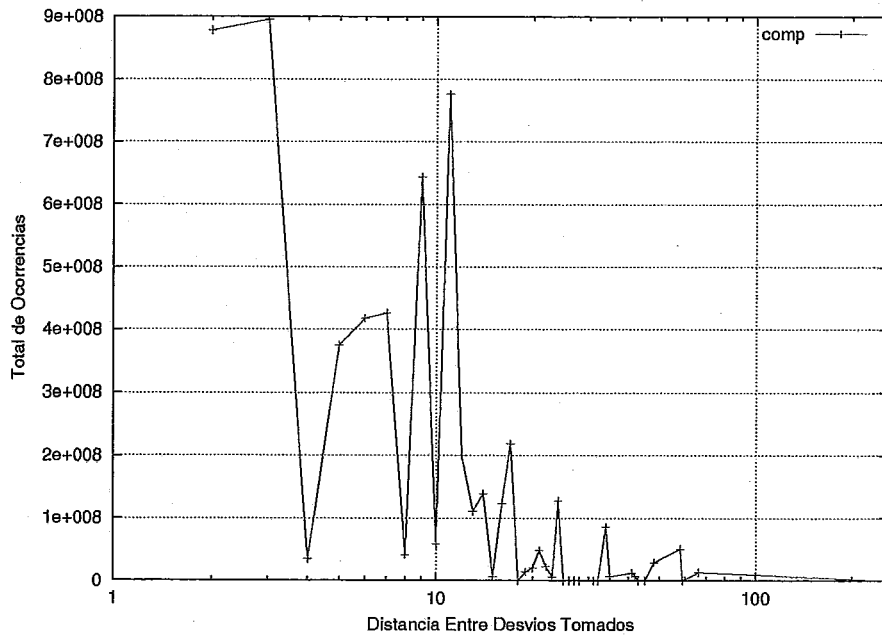


Figura 5.2: Distância Entre Desvios Tomados - compress e vortex

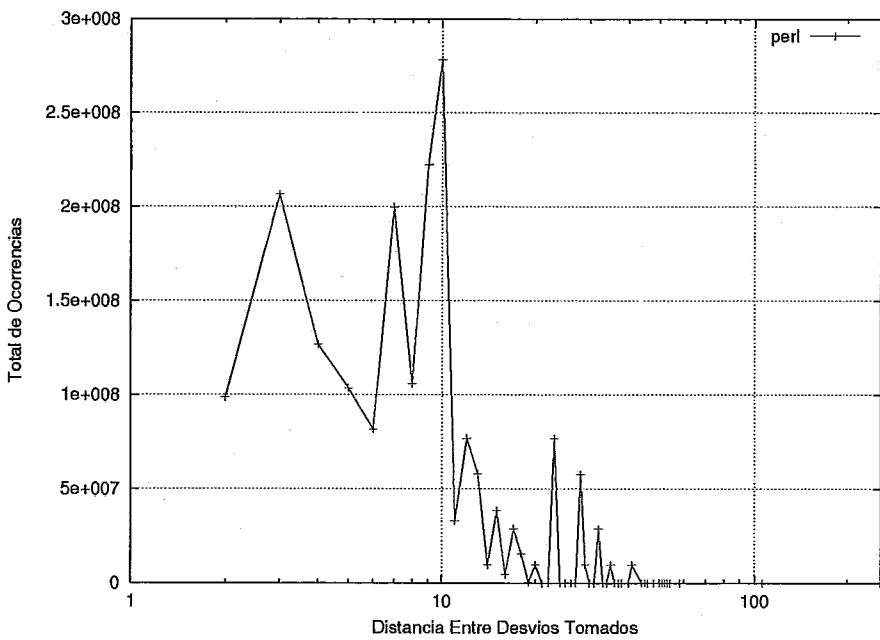
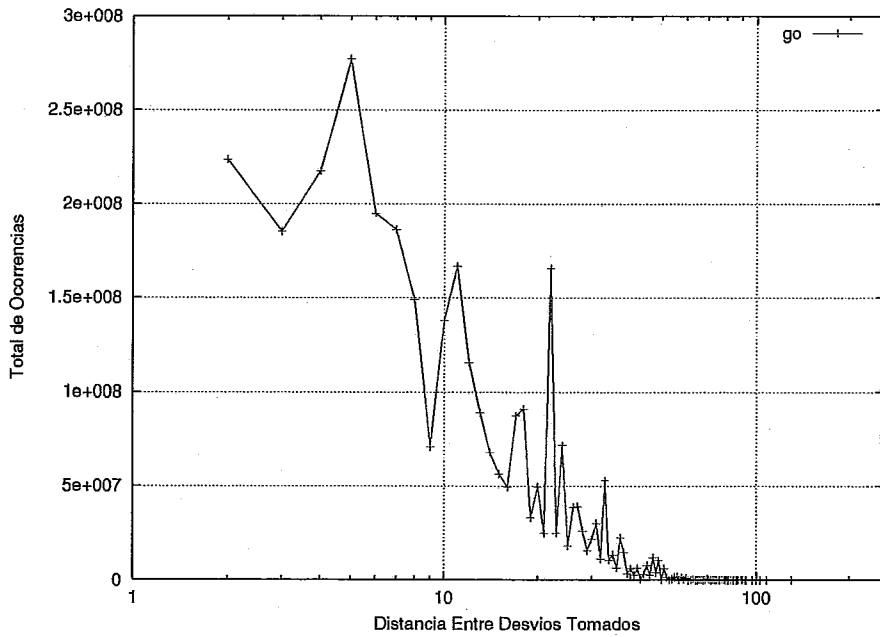


Figura 5.3: Distância Entre Desvios Tomados - go e perl



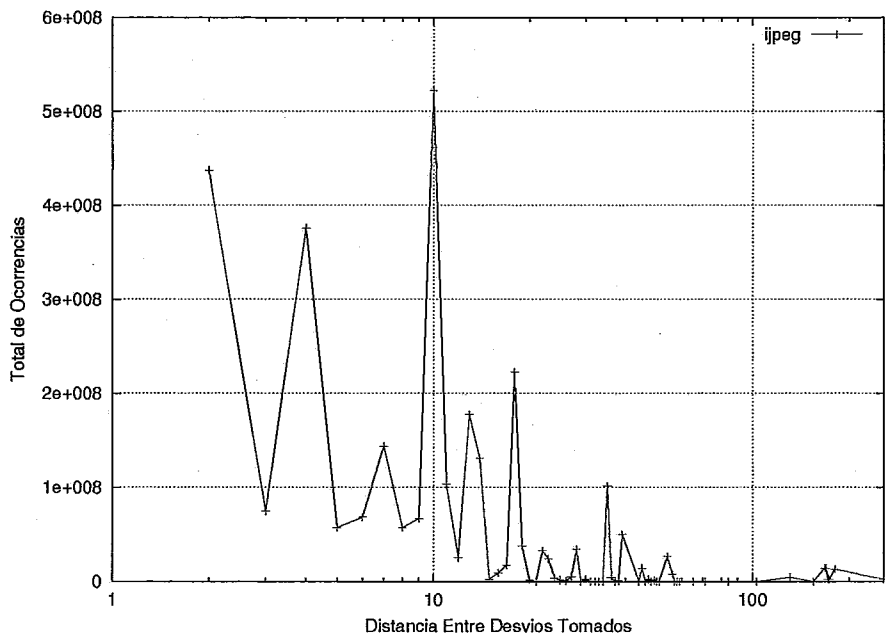
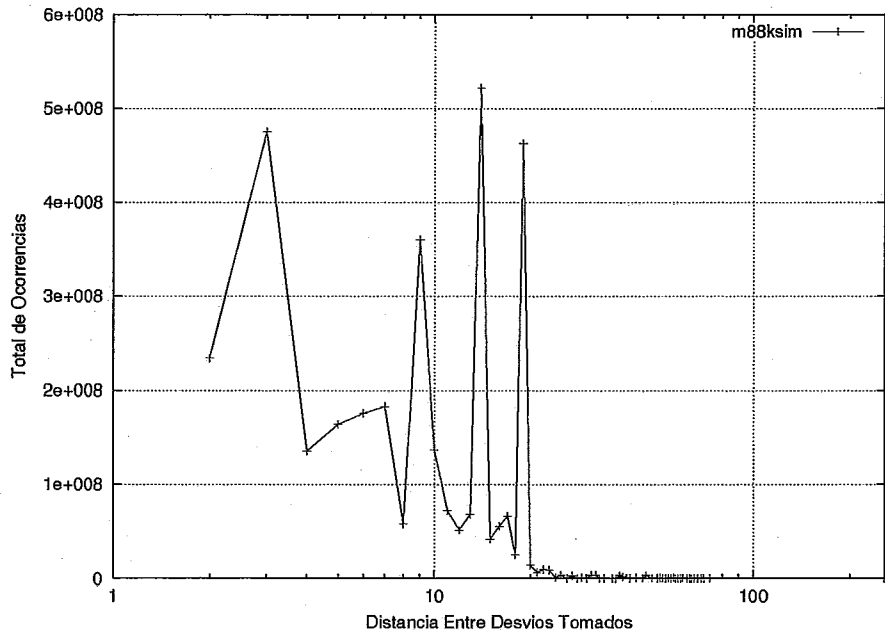


Figura 5.4: Distância Entre Desvios Tomados - m88ksim e jpeg

O programa *m88ksim* (Figura 5.4) apresenta moda igual a 14 instruções entre desvios. Distâncias com 3, 9 e 19 instruções também são muito frequentes. A ocorrência de distâncias com mais de 20 instruções não é significativa.

Finalmente, o programa *jpeg* (Figura 5.4) tem um comportamento similar ao do *m88ksim*, com moda igual a 10 instruções entre desvios, mas apresenta distâncias com até 256 instruções.

A principal conclusão que podemos obter dos gráficos é que: apesar dos programas do SPECint95 apresentarem um reduzido tamanho (estático) de bloco básico, durante a execução existe um número significativo de seqüências de instruções contíguas (sem desvios tomados), o que permite ganhos consideráveis no tempo de execução, se convenientemente exploradas.

### 5.2.3 Desvios e Blocos Básicos Estáticos de Maior Peso

Examinando o perfil de execução de um programa podemos observar que uma boa parte de suas instruções nunca são executadas [74]. O mesmo ocorre com seus blocos básicos. Para avaliar a percentagem de blocos básicos e de instruções de desvio do código objeto que são responsáveis pelo total (e por 90%) das instruções executadas conduzimos uma outra série de experimentos. Os dados por eles produzidos são **importantes parâmetros arquiteturais e não podem ser ignorados durante o dimensionamento das estruturas de armazenamento do nosso modelo básico de arquitetura.**

Nessa série de experimentos, monitoramos a execução dos blocos básicos dos programas inteiros do SPEC95. No final da execução de cada programa, verificamos quantas vezes cada bloco básico foi executado e qual a sua contribuição no total geral de instruções que o programa executou.

Os totais parciais de instruções executadas por cada bloco foram classificados em ordem decrescente e em seguida determinamos quantos blocos básicos foram responsáveis por 90 e 100% das instruções executadas pelo programa (segunda coluna da Tabela 5.2).

Na segunda parte dos experimentos, monitoramos as instruções de desvio. Para cada desvio executado, contamos o número de desvios condicionais e incondicionais. No término da simulação, avaliamos quantas instruções de desvio distintas foram executadas e dentre elas, quantas foram responsáveis por 90% das instruções de

Aplicação	Blocos Básicos	Desvios	Desvios Cond.	Desvios Incond.
go 100%	10.838	6.432	5.765	667
90%	1.062	781	753	28
gcc 100%	33.569	21.904	18.595	3.309
90%	4.003	3.103	2.799	304
m88k 100%	2.393	1.522	1.243	279
90%	187	141	129	12
comp 100%	707	431	354	77
90%	39	25	24	1
li 100%	1.883	971	800	171
90%	103	59	51	8
jpeg 100%	2.672	1.553	1.252	301
90%	50	36	29	7
perl 100%	2.903	1.815	1.478	337
90%	407	354	311	43
vortex 100%	12.931	7.670	6.548	1.122
90%	535	356	329	27

Tabela 5.2: Desvios e Blocos Básicos Executados nos Programas

desvio executadas. As colunas 3, 4 e 5 da Tabela 5.2 apresentam esses totais.

Examinando a Tabela 5.2, podemos observar o número de blocos básicos e desvios estáticos que participaram de 90% do total das instruções de desvio executadas. O número de blocos básicos que participa de 90% da computação é bastante reduzido para a maioria dos programas do SPECint95. Os programas que possuem um número significativo de desvios são o *gcc* e *go*. Os programas *li*, *jpeg* e *compress* possuem um número muito pequeno de desvios que são responsáveis pela maior parte da execução.

Ao contrário do que parecia inicialmente, o número de desvios que necessitam ser armazenados nos dispositivos de predição de desvio é bem menor, pois grande parte dos desvios estáticos é executada poucas ou nenhuma das vezes [74].

Esta característica facilita também a elaboração de um dispositivo para armazenamento dos blocos básicos mais executados. Pelos dados acima apresentados e com base também em outros trabalhos que investigam o comportamento dos blocos básicos [77], estimamos que uma *cache* com 2K entradas é suficiente para armazenar a maior parte dos blocos básicos executados.

Aplicação	Desvios Condicionais	Não Tomados	Tomados	Tomados p/ o Mesmo BB	Tomados p/ Alvo Último Desvio
go	92.852	22.139	70.713	302	0,7
m88ksim	21.142	9.516	11.625	2.332	4.123
gcc	264.076	118.351	145.725	26.941	32.982
compress	37.729	12.453	25.276	1.700	1.700
li	30.285	15.517	14.770	514	44
jpeg	140.548	41.300	99.248	9.722	9.768
perl	387.939	136.652	251.287	54.375	55.030
vortex	350.192	150.270	199.921	39.360	40.244

Tabela 5.3: Análise dos Desvios Condicionais (em milhares)

### 5.2.4 Localidade Espacial e Temporal

Outros experimentos para determinar a localidade dos programas do SPEC95 foram realizados. Estamos interessados em conhecer a localidade temporal e espacial que pode ser explorada pelas instruções na vizinhança do bloco básico que está sendo executado. A Tabela 5.3 mostra, para cada aplicação, o total de desvios condicionais, quantos não foram tomados e quantos foram tomados. A penúltima coluna mostra quantos desvios foram tomados para o mesmo bloco básico, isto é, quantos desvios cujo endereço alvo era igual ao endereço da primeira instrução do bloco básico. A última coluna diz quantos desvios tomados tinham endereço alvo igual ao endereço alvo do último desvio tomado. Os dados foram obtidos durante a execução dos programas com a entrada *training* do SPEC95.

Note que os dados da última e penúltima coluna não são exclusivos, ou seja, a intersecção dos conjuntos de dados não é vazia. Os dados indicam que uma boa parte dos desvios tomados ou é para o mesmo bloco básico ou para o endereço alvo do último desvio tomado. As exceções são os programas *li* e *go*, provavelmente pelo grande número de pequenas rotinas que são chamadas e executadas nesses programas.

Esta localidade indica que podemos empregar uma pequena memória *cache* associativa para fornecer as instruções mais recentemente utilizadas, eventualmente já decodificadas, sem a complexidade do mecanismo *trace cache*.

A Tabela 5.4 apresenta o total de instruções executadas por cada aplicação com a entrada *training* e a taxa de acerto das instruções em relação ao último *buffer* despa-

Aplicação	Total de Instruções	Taxa de Acerto <i>Buffer</i> de Despacho
go	1.171.408.043	3,1%
m88ksim	148.304.898	33,0%
gcc	1.480.605.365	18,5%
compress	371.253.220	18,6%
li	236.039.368	8,0%
ijpeg	2.110.119.608	11,5%
perl	2.331.095.615	22,8%
vortex	2.426.622.809	7,7%

Tabela 5.4: Taxa de Acerto no Último *Buffer* de Despacho

chado. Ou seja, estamos verificando quantas instruções que estão sendo despachadas neste ciclo, que foram despachadas no ciclo imediatamente anterior.

Nesse experimento, consideramos uma arquitetura equipada com um *buffer* de 16 instruções e com predição de desvios onisciente. Embora o despacho de várias instruções por ciclo tenha uma série de desvantagens, o grande número de instruções despachadas em paralelo apresenta outras características favoráveis. Os resultados mostram que é possível reaproveitar as instruções que acabaram de ser despachadas, diminuindo assim a pressão sobre os mecanismos de busca.

## 5.2.5 Comentários

Entre os resultados apresentados nesta seção, podemos destacar o pequeno número de blocos básicos que efetivamente realizam a computação e, conseqüentemente, o pequeno número de desvios distintos que são executados ao longo dos programas.

No programa *compress*, por exemplo, apenas 39 blocos básicos distintos são responsáveis por 90% das instruções executadas e apenas 25 desvios distintos fazem parte de 90% do fluxo de instruções executadas. Isto explica, por exemplo, os resultados favoráveis obtidos com o uso de *caches* de instrução, *trace cache* e mecanismos de busca de instruções.

Um tamanho de linha de *cache* para atender tanto às necessidades dos programas inteiros como de ponto flutuante, seria algo em torno de 14 instruções, se tivermos uma largura de despacho correspondente. Para larguras de despacho menores, em torno de 8 instruções, uma linha de *cache* do mesmo tamanho atenderia convenien-

temente os dois tipos de programas.

As taxas de acerto referentes ao último *buffer* despachado, assim como os desvios tomados para o mesmo bloco básico ou de volta para o último desvio tomado sugerem o desenvolvimento de uma pequena *cache* associativa, ou mesmo reutilizar as instruções que estiverem na janela de instruções, diminuindo assim a pressão sobre os mecanismos de busca.

As características observadas são importantes para a modelagem de preditores de múltiplos desvios e *caches* especiais para armazenamento de seqüências de *traces* de blocos básicos. Como visto, existe um grande potencial de paralelismo nos programas analisados, mesmo para os programas inteiros.

### 5.3 Avaliação do Despacho Seletivo

Aplicação	Total de Instruções
go	1.171.408.043
m88ksim	148.304.898
gcc	1.480.605.365
compress	371.253.220
li	236.039.368
jpeg	2.110.119.608
perl	2.331.095.615
vortex	2.426.622.809

Tabela 5.5: Total de Instruções Executadas - Prog. Inteiros

Nesta seção apresentamos os resultados da avaliação do mecanismo de despacho seletivo apresentado no capítulo 3. Para tal, simulamos os programas inteiros e de ponto flutuante do SPEC95, com a entrada de dados *training*.

Inicialmente apresentaremos um perfil da execução dos programas do SPEC95, listando o total dinâmico de instruções com 0, 1, 2 e 3 dependências de dados. O somatório destes valores é igual ao total de instruções executadas por cada programa com a entrada *training*, que pode ser visto nas Tabelas 5.5 e 5.6.

Na arquitetura SPARC, as instruções podem ter até dois registradores arquiteturais e o código de condição como operandos fonte. Cada um deles depende do resultado de alguma instrução executada anteriormente, que pode estar ou não no mesmo

Aplicação	Total de Instruções
tomcatv	8.196.055.564
swim	436.838.275
su2cor	8.492.129.575
hydro2d	7.535.040.453
mgrid	8.158.173.868
applu	555.460.474
turb3d	17.488.499.204
apsi	2.528.799.740
fpppp	479.939.762
wave5	3.607.930.647

Tabela 5.6: Total de Instruções Executadas - Ponto Flutuante

bloco básico. Instruções com 3 dependências lêem dois operandos em registradores (não-imediatos) e usam o código de condição. Instruções com 2 dependências ou lêem dois operandos não-imediatos ou então um operando não-imediato e o código de condição. E assim por diante. Existem ainda instruções que não requerem comparações para serem despachadas, estas são contabilizadas como “0” dependências.

Oper. Não-Imed.	3	2	1	0
<b>Aplicação</b>				
go	17.109.116	205.770.425	694.216.563	254.311.939
m88ksim	284.674	29.664.222	99.865.063	18.490.941
gcc	7.835.554	219.208.295	1.102.524.347	151.037.189
compress	4.869.217	96.497.502	211.440.438	58.446.063
li	373.635	34.927.468	178.896.738	21.839.384
ijpeg	397.565.328	647.612.025	975.367.267	89.543.244
perl	23.866.405	424.694.267	1.602.073.258	280.461.649
vortex	3.994.548	331.545.779	1.599.200.409	491.882.713

Tabela 5.7: Total de Instruções por Número de Dependências - Prog. Inteiros

Nas Tabelas 5.7 e 5.8 são mostrados os totais de instruções organizados em colunas, de acordo com o número de operandos não-imediatos (ou dependências) de cada instrução, respectivamente, para os programas inteiros e de ponto flutuante do SPEC95.

Nas Tabelas 5.9 e 5.10 os mesmos resultados são listados, mas desta vez em valores percentuais em relação ao total de instruções executadas em cada programa.

Para cada operando que a instrução possua, existe uma dependência verdadeira

Oper. Não-Imed.	3	2	1	0
Aplicação				
tomcatv	1.207.834	4.024.662.272	4.011.639.228	158.546.230
swim	30.943.542	292.031.205	94.054.369	19.809.159
su2cor	39.360.806	4.240.661.873	3.816.795.056	395.311.840
hydro2d	434.454	2.853.847.156	4.047.849.859	632.908.984
mgrid	11.849.903	3.512.511.648	4.623.735.857	10.076.460
applu	57.805	208.178.531	342.720.092	4.504.046
turb3d	303.099.954	8.083.695.410	8.586.536.425	515.167.415
apsi	58.236.18	1.118.047.396	1.277.294.321	75.221.842
fpppp	566.692	162.782.548	313.470.936	3.119.586
wave5	75.037.426	1.606.938.991	1.762.690.906	163.263.324

Tabela 5.8: Total de Instruções por Número de Dependências - Ponto Flutuante

com uma instrução precedente. É possível que algumas dessas instruções precedentes estejam sendo despachadas simultaneamente e por esse motivo, a instrução deve esperar que os resultados sejam produzidos pela instrução predecessora, antes que possa ser executada.

Se a instrução não possuir operandos fonte, ela pode ser despachada imediatamente; caso possua 1 ou mais operandos, é necessário verificar se existe dependência direta com alguma outra instrução que está sendo despachada naquele ciclo. Para isto será necessário realizar comparações com todas as instruções precedentes que estiverem sendo despachadas simultaneamente.

Conforme podemos observar na Tabela 5.9 durante a execução dos programas inteiros, ocorreu um elevado número de instruções independentes: de 4,2% (*jpeg*) a 21,7% (*go*) das instruções podem ser despachadas a cada ciclo sem comparações, ou seja, podem ser enviadas para um estação de reserva sem comparadores.

A maior parte das instruções dos programas inteiros possui apenas um operando de leitura. Isso pode ser constatado pelo elevado percentual de instruções com apenas uma dependência, que varia entre 46,2% (*jpeg*) e 75,8% (*li*). O percentual de instruções com dois operandos de leitura varia entre 14,8% (*gcc* e *li*) e 30,7% (*jpeg*). Finalmente, o menor percentual encontrado é o de instruções com 3 operandos de leitura, variando entre 0,1% (*li*) e 1,5% (*go*). A exceção é o programa *jpeg* com 18,8% das instruções com 3 dependências.

Os programas de ponto flutuante possuem um número menor de instruções in-



Oper. Não-Imed.	3	2	1	0
Aplicação				
go	1,5	17,5	59,3	21,7
m88ksim	0,2	20,0	67,5	12,5
gcc	0,5	14,8	74,5	10,2
compress	1,3	26,0	57,0	15,8
li	0,1	14,8	75,8	9,3
jpeg	18,8	30,7	46,2	4,2
perl	1,0	18,2	68,7	12,0
vortex	0,2	13,7	65,9	20,3

Tabela 5.9: Percentual por Número de Dependências - Prog. Inteiros

Oper. Não-Imed.	3	2	1	0
Aplicação				
tomcatv	0,0	49,1	48,8	2,0
swim	6,9	67,0	21,6	4,3
su2cor	0,5	49,9	45,0	4,6
hydro2d	0,1	37,9	53,7	8,4
mgrid	0,1	43,0	56,7	0,1
applu	0,1	37,5	61,6	0,7
turb3d	1,7	46,2	49,1	2,9
apsi	2,3	44,2	50,5	2,9
fpppp	0,3	33,8	65,3	0,6
wave5	2,1	44,5	48,8	4,5

Tabela 5.10: Percentual Instruções por Número de Dependências - Ponto Flutuante

dependentes: entre 0,1% (*mgrid*) e 8,4% (*hydro2d*). De uma maneira geral, a maior parte das instruções possui um operando fonte, entre 21,6% (*swim*) e 65,33% (*fpppp*). Contudo, o percentual de instruções com dois operandos fonte também é alto: entre 33,8% (*fpppp*) e 67,0% (*swim*). Assim como nos programas inteiros, o percentual das instruções com 3 dependências é bastante reduzido: entre 0% (*tomcatv*) e 2,3% (*apsi*). A exceção fica por conta do programa *swim* com 6,9% das instruções com 3 operandos fonte.

As percentagens de instruções com três operandos fonte são muito baixas, e por esse motivo, nosso mecanismo de despacho seletivo torna-se vantajoso: ao dispensar comparadores e barramentos para a recepção de resultados produzidos por outras instruções, estações de reserva especializadas tornam nosso modelo de máquina mais simples e eficiente. Por exemplo, podemos verificar que entre 50% (*jpeg*) e 90%

(*m88ksim*) das instruções dos programas inteiros precisam de estações de reserva com no máximo um comparador por barramento de resultado (CDB). Para os programas de ponto flutuante, com exceção do programa *swim*, entre 49,6% e 65,9% das instruções despachadas possuem 1 ou menos operandos fonte.

Este perfil de execução indica que o número de comparadores necessários para o despacho de múltiplas instruções em um programa de ponto flutuante é maior do que em programas inteiros.

Na seção seguinte analisamos os resultados dos experimentos com os três tipos de *cache* de dependências.

## 5.4 Avaliação da *Cache* de Dependências

Nesta seção avaliamos a eficiência dos três modelos de *cache* de dependências do nosso modelo básico. O desempenho das arquiteturas com esses três modelos de *cache* é comparado com o desempenho de uma arquitetura usada como referência e que é derivada do modelo básico, mas que não foi equipada com *cache* de dependências.

Nos gráficos e tabelas apresentados, a máquina de referência é indicada pela sigla **DE**. Quando a arquitetura estiver equipada com uma *cache* de dependências simples, isto será expresso pela sigla **CS**. No caso da *cache* de dependências inteligente a sigla utilizada é **CI**. Finalmente, a sigla **CA** indica uma arquitetura que faz uso da *cache* de dependências avançada. Nas tabelas apresentadas, antes de cada sigla, um algarismo indica a largura de despacho da arquitetura simulada (8 ou 16 instruções).

A arquitetura de referência (DE) detecta as dependências comparando os operandos fonte de cada instrução com os operandos destino de todas as instruções precedentes que estão sendo despachadas em paralelo. O número de comparações necessárias ao despacho de cada instrução leva em conta o número de operandos não-imediatos que a instrução possui. Por exemplo, instruções que possuam apenas operandos imediatos são despachadas sem comparações. Ou seja, a nossa arquitetura de referência já possui otimizações que reduzem o número de comparações para o despacho de cada instrução.

Considerando que nossa arquitetura pode ter até três operandos por instrução, o valor máximo teórico para o número de comparações necessárias para despachar

corretamente as instruções no *buffer* de despacho, seria de

$$\frac{3 \times N \times (N - 1)}{2}$$

ou seja, 84 comparações para uma largura de despacho com 8 instruções, ou 360 comparações para uma largura de despacho com 16 instruções.

No restante desta seção, descreveremos os experimentos com os programas inteiros e de ponto flutuante do SPEC95, com a entrada de dados *training* e que foram simulados em máquinas experimentais que despacham 8 e 16 instruções por ciclo.

### 5.4.1 Largura de Despacho de 8 Instruções

Descrevemos a seguir os resultados obtidos com a simulação de uma arquitetura com largura de despacho com 8 instruções. Primeiramente, apresentamos os resultados obtidos com a simulação dos programas inteiros e, logo a seguir, com os programas de ponto flutuante do SPEC95.

Despacho	8-DE	8-CS	8-CI	8-CA
<b>Aplicação</b>				
go	4.049	1.112	561	86
m8ksim	559	315	194	50
gcc	5.475	2.964	1.999	537
compress	1.466	623	336	79
li	874	393	245	50
jpeg	12.121	5.474	2.351	456
perl	8.831	5.220	3.547	940
vortex	7.959	4.143	2.684	642

Tabela 5.11: Comparações (em milhões) - Prog. Inteiros

A Tabela 5.11 apresenta o total de comparações requerido para o despacho de *todas* as instruções executadas nos programas inteiros do SPEC95. A segunda coluna apresenta os resultados da simulação da arquitetura de referência (DE). As três últimas colunas apresentam o total de comparações necessárias para o despacho dos programas, utilizando cada um dos modelos de *cache* de dependências. Na coluna 3 estão os resultados da *cache* simples (CS), na coluna seguinte os totais para a *cache* inteligente (CI) e na última coluna os totais para a *cache* avançada (CA). O

número de comparações é proporcional ao total de instruções executadas e podemos observar que ele diminui conforme aumenta a sofisticação do modelo de *cache* de dependências.

Despacho	8-DE	8-CS	8-CI	8-CA
<b>Aplicação</b>				
go	3,5	1,0	0,5	0,1
m88ksim	3,8	2,1	1,3	0,3
gcc	3,7	2,0	1,4	0,4
compress	3,9	1,7	0,9	0,2
li	3,7	1,7	1,0	0,2
jpeg	5,7	2,6	1,1	0,2
perl	3,8	2,2	1,5	0,4
vortex	3,3	1,7	1,1	0,3

Tabela 5.12: Comparações por Instrução Despachada - Prog. Inteiros

Na Tabela 5.12 apresentamos o número médio de comparações necessários ao despacho de cada instrução. Este valor foi obtido dividindo-se o total de comparações exibido na Tabela 5.11 pelo total de instruções executadas por cada programa. Na Figura 5.5 os mesmos valores são apresentados na forma de um gráfico.

Estes valores indicam o número mínimo de comparadores que devemos ter disponíveis na arquitetura. Por exemplo, para uma arquitetura sem *cache* de dependências, seriam necessários no mínimo  $5,7 \times 8 = 46$  comparadores para despachar convenientemente as instruções. Com o uso de uma *cache* simples o número necessário cai para cerca de 21 comparadores. No caso de uma *cache* inteligente seriam 12 comparadores ou 4 comparadores com o uso da *cache* avançada.

Estes valores servem apenas como referência, pois na prática valores maiores seriam necessários, para atender os casos onde a composição das instruções armazenadas no *buffer* de despacho necessitasse de um número de maior de comparações. Lembramos que o máximo teórico é de 84 comparadores para uma largura de despacho com 8 instruções.

A Tabela 5.13 apresenta os mesmos dados em forma percentual. A arquitetura de referência (DE) foi utilizada como padrão de comparação, ou seja, todas as colunas mostram, em valores percentuais, o número de comparações em cada modelo de *cache* dividido pelo total de comparações obtido na arquitetura de referência.

Com o uso da *cache* simples, o número de comparações foi reduzido para 46,6%

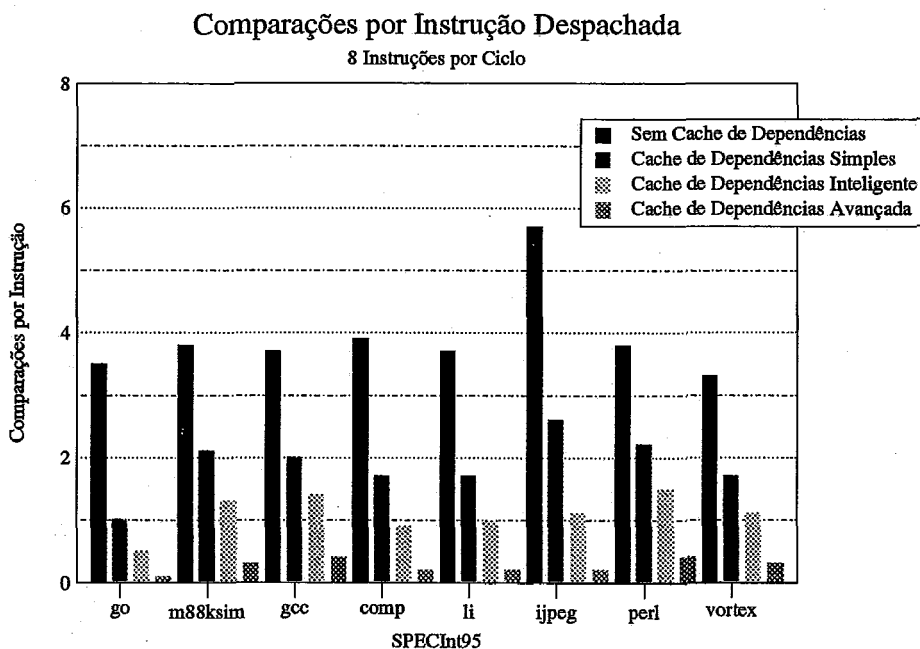


Figura 5.5: Comparações por Instrução Despachada - *Buffer c*/ 8 Instruções

do total obtido nas simulações da arquitetura de referência. Para a arquitetura equipada com a *cache* inteligente o número de comparações reduziu-se a 27,2% ou para apenas 2,9% quando uma *cache* avançada estava em uso.

Estes valores mostram uma redução no número de comparações para todos os modelos de *cache*, sendo que a *cache* avançada apresentou resultados significativamente melhores.

Na Tabela 5.14 encontramos os totais de comparações necessárias para o despacho dos 10 programas de ponto flutuante do SPEC95. O uso da *cache* de dependências propicia uma grande redução no número de comparações necessárias para o despacho das instruções.

Na divisão do total de comparações pelo total de instruções de cada programa, obtivemos os dados apresentados na Tabela 5.15, que indicam o número médio de comparações por instrução para o despacho dos programas de ponto flutuante.

Os programas de ponto flutuante exigem um número maior de comparadores na arquitetura para o despacho das instruções. Conforme visto nas Tabelas 5.9 e 5.10, existe um percentual maior de instruções com 2 ou 3 operandos do que nos programas inteiros, o que gera mais dependências com outras instruções. Deste modo, o número mínimo de comparadores para o despacho de todas as instruções no mesmo *buffer*

Despacho	8-CS	8-CI	8-CA
Aplicação			
go	27,5	13,9	1,0
m88ksim	56,5	34,7	4,2
gcc	54,1	36,5	4,6
compress	42,5	23,0	2,5
li	45,0	28,1	2,7
ijpeg	45,2	19,4	1,8
perl	59,1	40,2	5,0
vortex	52,1	33,7	3,8
MÉDIA	46,6	27,2	2,9

Tabela 5.13: Percentual de Comparações - Prog. Inteiros

Desp.	8-DE	8-CS	8-CI	8-CA
Aplic.				
tomcatv	46.345	28.721	8.961	1.285
swim	2.698	1.604	482	91
su2cor	83.682	54.872	18.368	2.964
hydro2d	34.145	23.888	9.406	1.858
mgrid	88.325	43.843	14.877	1.763
applu	2.657	1.610	515	94
turb3d	89.820	61.264	19.289	3.829
apsi	12.908	7.851	2.735	492
fpppp	2.242	1.194	372	48
wave5	18.211	10.372	3.548	627

Tabela 5.14: Comparações (em milhões) - Ponto Flutuante

de despacho (DE) seria de 50 comparadores na arquitetura de referência. Com o uso da *cache* de dependências, este número cairia para 30, 10 e 2 comparadores, com o uso dos modelos simples, inteligente e avançado, respectivamente.

Os resultados da Tabela 5.15 são apresentados em forma de gráfico na Figura 5.6.

A Tabela 5.16 apresenta, percentualmente, o total de comparações para cada modelo de *cache* de dependências, relativamente ao total de comparações medido na arquitetura de referência (DE), com largura de despacho de 8 instruções.

Com o uso da *cache* simples o número médio de comparações reduz-se a 60,3% do total requerido na arquitetura de referência. A *cache* inteligente permite uma redução média para 19,8% das comparações, enquanto que apenas 1,5% com o uso da *cache* avançada.

Despacho	8-DE	8-CS	8-CI	8-CA
Aplicação				
tomcatv	5,2	3,2	1,0	0,1
swim	6,2	3,7	1,1	0,2
su2cor	5,2	3,4	1,0	0,2
hydro2d	4,5	3,2	1,2	0,2
mgrid	5,0	2,5	0,8	0,1
applu	4,8	2,9	0,9	0,2
turb3d	5,1	3,5	1,1	0,2
apsi	5,1	3,1	1,1	0,2
fpppp	4,7	2,5	0,8	0,1
wave5	5,0	2,9	1,0	0,2

Tabela 5.15: Comparações por Instrução Despachada - Ponto Flutuante

Despacho	8-CS	8-CI	8-CA
Aplicação			
tomcatv	62,0	19,3	1,3
swim	59,4	17,9	1,6
su2cor	65,6	19,6	1,7
hydro2d	70,0	27,5	2,5
mgrid	49,6	16,8	0,9
applu	60,6	19,4	1,7
turb3d	68,2	21,5	2,0
apsi	60,8	21,2	1,8
fpppp	53,3	16,6	1,0
wave5	57,0	19,5	1,6
MÉDIA	60,3	19,8	1,5

Tabela 5.16: Percentual de Comparações - Ponto Flutuante

### 5.4.2 Largura de Despacho de 16 Instruções

Na seção anterior avaliamos uma arquitetura com largura de despacho de 8 instruções. Agora estamos interessados em observar o comportamento de nossos modelos de *cache* de dependências em uma arquitetura com largura de despacho de 16 instruções, onde o número de comparadores necessários para o despacho das instruções é bem maior que no caso anterior.

Do mesmo modo que foi feito anteriormente, simulamos primeiramente os programas inteiros e a seguir os de ponto flutuante.

Na Tabela 5.17 estão os resultados da simulação dos programas inteiros do SPEC95 nas máquinas com os três modelos de *cache* de dependências e na ar-

## Comparações por Instrução Despachada

8 Instruções por Ciclo

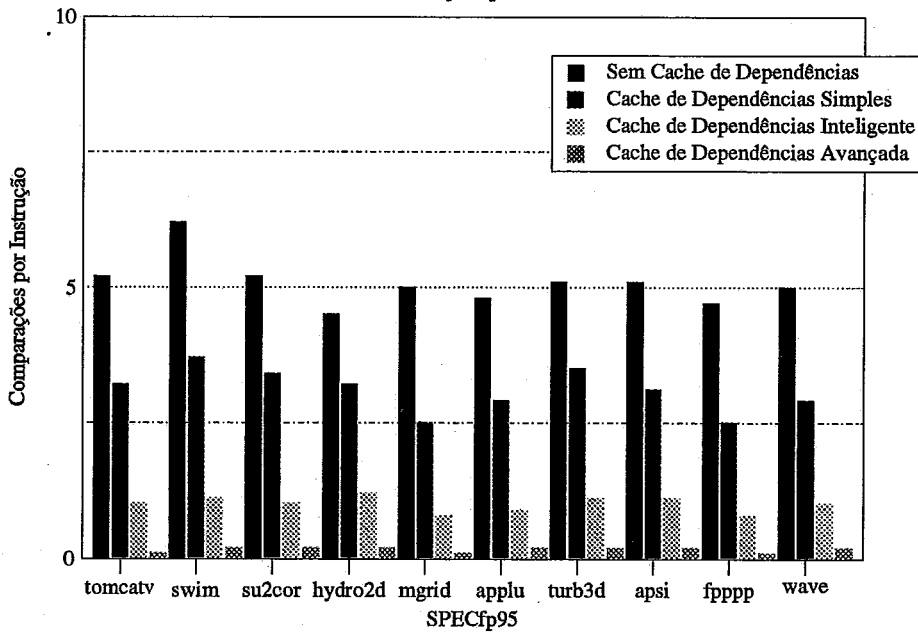


Figura 5.6: Comparações por Instrução Despachada - *Buffer c*/ 8 Instruções

quietura de referência. Em cada coluna está o total de comparações necessárias ao despacho de todas as instruções dos programas inteiros do SPEC95, em uma máquina com largura de despacho de 16 instruções.

Conforme aumenta o total de instruções, aumenta o total de comparações requeridas para o despacho das instruções executadas em cada um dos programas. Podemos verificar, entretanto, que estes totais são reduzidos significativamente quando utilizamos os modelos de *cache* de dependências.

Na Tabela 5.18 é listado o número médio de comparações que são realizadas para o despacho de cada instrução dos programas do SPECint95. Este valor dobrou se comparado com os resultados obtidos com uma largura de 8 instruções (veja Tabela 5.12), conforme era esperado.

Para esta largura de despacho, são necessários um mínimo de  $12,3 \times 16 = 197$  comparadores para o correto despacho das instruções sem o uso da *cache* de dependências. Se a arquitetura for modificada para o uso de uma *cache* de dependências simples, seriam necessários 90 comparadores. Para o caso de utilizarmos uma *cache* inteligente este valor se reduz para 63 comparadores. Finalmente, com a *cache* avançada, temos a necessidade do uso de apenas 15 comparadores. O máximo



<b>Despacho</b>	16-DE	16-CS	16-CI	16-CA
<b>Aplicação</b>				
go	8.678	2.384	1.570	210
m88ksim	1.199	678	509	108
gcc	11.733	6.352	5.091	1.178
compress	3.142	1.334	949	176
li	1.874	842	654	122
ijpeg	25.974	11.730	6.761	1.057
perl	18.922	11.187	9.130	2.110
vortex	17.056	8.879	7.019	1.448

Tabela 5.17: Comparações (em milhões) - Prog. Inteiros

téorico calculado é de 360 comparadores.

<b>Despacho</b>	16-DE	16-CS	16-CI	16-CA
<b>Aplicação</b>				
go	7,4	2,0	1,3	0,2
m88ksim	8,1	4,6	3,4	0,7
gcc	7,9	4,3	3,4	0,8
compress	8,5	3,6	2,6	0,5
li	7,9	3,6	2,8	0,5
ijpeg	12,3	5,6	3,2	0,5
perl	8,1	4,8	3,9	0,9
vortex	7,0	3,7	2,9	0,6

Tabela 5.18: Comparações por Instrução Despachada - Prog. Inteiros

O gráfico na Figura 5.7 apresenta a média de comparações para o despacho de cada instrução dos programas inteiros do SPEC95, nas simulações em arquiteturas com largura de despacho de 16 instruções.

A Tabela 5.19 apresenta o percentual do total de comparações para cada um dos modelos de *cache* em relação à arquitetura de referência (DE), considerando-se uma largura de despacho de 16 instruções.

Com o uso da *cache* simples, o número de comparações caiu para um valor que é 46,6% do valor da arquitetura de referência. Para as *caches* inteligente e avançada os valores médios se situam em 34,1% e 6,5%, respectivamente, do obtido com a arquitetura de referência.

Novamente obtivemos as maiores reduções com o uso da *cache* avançada.

Os dados a seguir apresentam os resultados obtidos para a simulação dos pro-

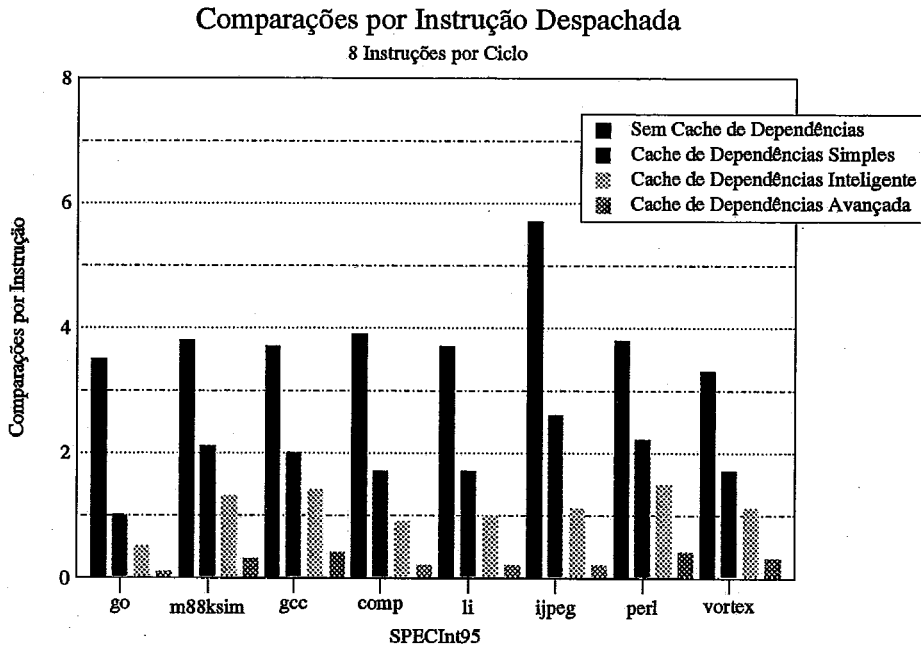


Figura 5.7: Comparações por Instrução Despachada - *Buffer c/ 16 Instruções*

gramas de ponto flutuante com os três modelos de cache de dependência.

O mesmo comportamento se observa na Tabela 5.20 em relação à arquitetura com largura de despacho de 8 instruções: os valores dobram, pois a largura de despacho também dobrou.

Se dividirmos o total de comparações realizadas pelo número de instruções executadas em cada programa, encontraremos os resultados que são apresentados na Tabela 5.21, que é o número médio de comparações necessários para o despacho de cada instrução.

A partir deste valor, obtemos indicações do número de comparadores para o despacho simultâneo de 16 instruções, quando executando programas de ponto flutuante: 211 comparadores na arquitetura de referência; que poderiam ser reduzidos para 126 com uso da *cache* simples; ou 63 com a *cache* inteligente; ou ainda 10 comparadores empregando-se a *cache* avançada. O número máximo teórico de comparadores que seriam necessários é de 360.

Comparando-se com os programas inteiros, surpreendentemente, os esquemas de *cache* inteligente e avançada são mais eficientes. Isto deve, muito provavelmente, ao menor número de blocos básicos presentes em cada *buffer* de despacho.

O gráfico na Figura 5.8 apresenta os resultados da Tabela 5.21 para os programas

Despacho	16-CS	16-CI	16-CA
Aplicação			
go	27,5	18,1	2,4
m88ksim	56,5	42,5	9,1
gcc	54,1	43,4	10,0
compress	42,5	30,2	5,6
li	45,0	34,9	6,5
ijpeg	45,2	26,0	4,1
perl	59,1	48,3	11,2
vortex	52,1	41,2	8,5
MÉDIA	46,6	34,1	6,5

Tabela 5.19: Percentual de Comparações - Prog. Inteiros

Desp.	16-DE	16-CS	16-CI	16-CA
Aplic.				
tomcatv	99.314	61.546	29.907	3.199
swim	5.781	3.437	1.625	204
su2cor	179.318	117.582	56.081	6.995
hydro2d	73.173	51.190	29.297	4.186
mgrid	189.270	93.951	47.826	4.264
applu	5.694	3.451	1.716	215
turb3d	192.474	131.280	63.826	8.701
apsi	27.660	16.825	8.712	1.129
fpppp	4.805	2.559	1.245	116
wave5	39.018	22.224	11.466	1.459

Tabela 5.20: Comparações (em milhões) - Ponto Flutuante

de ponto flutuante do SPEC95.

Em termos percentuais, como mostrado na Tabela 5.22, os modelos de *cache* de dependências apresentam os seguintes resultados: a *cache* simples realiza 60,3% das comparações realizadas na arquitetura de referência. As *caches* inteligente e avançada apresentam valores de 30,3% e 3,6% daqueles da arquitetura de referência, com uma largura de despacho de 16 instruções e executando os programas de ponto flutuante do SPEC95.

Despacho	16-DE	16-CS	16-CI	16-CA
Aplicação				
tomcatv	11,1	6,8	3,3	0,4
swim	13,2	7,9	3,7	0,5
su2cor	11,1	7,3	3,5	0,4
hydro2d	9,7	6,8	3,9	0,6
mgrid	10,7	5,3	2,7	0,2
applu	10,3	6,2	3,1	0,4
turb3d	11,0	7,5	3,6	0,5
apsi	10,9	6,7	3,4	0,4
fpppp	10,0	5,3	2,6	0,2
wave5	10,8	6,2	3,2	0,4

Tabela 5.21: Comparações por Instrução Despachada - Ponto Flutuante

Despacho	CS	16-CI	16-CA
Aplicação			
tomcatv	62,0	30,1	3,2
swim	59,4	28,1	3,5
su2cor	65,6	31,3	3,9
hydro2d	70,0	40,0	5,7
mgrid	49,6	25,3	2,3
applu	60,6	30,1	3,8
turb3d	68,2	33,2	4,5
apsi	60,8	31,5	4,1
fpppp	53,3	25,9	2,4
wave5	57,0	29,4	3,7
MÉDIA	60,3	30,3	3,6

Tabela 5.22: Percentual de Comparações - Ponto Flutuante

## 5.5 Resumo

Os programas do conjunto SPEC95 são amplamente difundidos e utilizados na modelagem de novas arquiteturas. Neste capítulo mostramos em detalhes as características desses programas, de modo a tornar claro os aspectos de uma arquitetura que podem ser convenientemente avaliados com seu uso.

Os programas inteiros que são mais significativos são o *go*, *gcc*, *perl* e *vortex*, porque apresentam um número relativamente grande de blocos básicos que participam da computação. Isto foi sempre levado em consideração, durante os estudos que realizamos.

No momento da redação desta tese, uma nova família de programas de avaliação

## Comparações por Instrução Despachada

16 Instruções por Ciclo

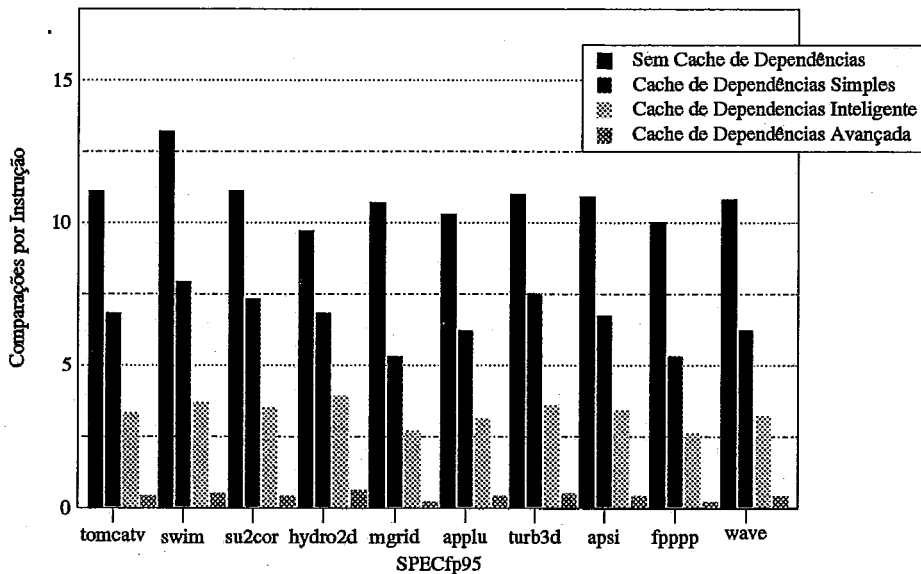


Figura 5.8: Comparações por Instrução Despachada - *Buffer c*/ 16 Instruções

foi lançada: o SPEC2000. Os programas do SPECint95 que ainda permaneceram nesta suíte foram o *gcc*, *vortex* e *perl*, embora com algumas modificações, quer no código, quer na entrada de dados. Estes fatos vêm comprovar a qualidade dos resultados apresentados nesta tese.

No levantamento das características de paralelismo de baixo nível nos programas inteiros do SPEC95 obtivemos dados muito interessantes. O número médio de instruções entre desvios **tomados** varia de 6,5 a 14,3, com média ponderada igual a 10,2. Estes valores são bem maiores que o número médio de instruções despachadas a cada ciclo pela arquiteturas atuais e permite uma conveniente extração de paralelismo, se pudermos prever pelo menos dois desvios com alto grau de acurácia. Neste sentido, podemos destacar o trabalho de Michaud et al. [30] em que é proposto um *extended two-block ahead predictor*, que faz a predição de até quatro blocos básicos, quando o primeiro e o terceiro são ambos não tomados.

Para os programas de ponto flutuante não realizamos o estudo para medir a distância entre os desvios tomados, pois a oportunidade de exploração de paralelismo existente já é bastante significativa dentro de um mesmo bloco básico.

Os estudos que realizamos levantaram características importantes dos programas SPEC95, muitas delas inéditas. Estes resultados permitem que a pesquisa se apro-

funde em várias direções. Optamos, no momento, por concentrar a nossa pesquisa na otimização do estágio de despacho, em particular em mecanismos que diminuam o *hardware* envolvido no despacho de múltiplas instruções.

Em relação ao despacho seletivo, os dados indicam que 50% das estações de reserva só necessitam ter um comparador, ao invés de três. Para uma janela de despacho com 64 instruções, por exemplo, isto significa uma economia significativa em termos de total de comparadores, registradores e de espaço gasto com o roteamento de barramento de resultados.

A *cache* avançada também pode ser utilizada para diminuir a complexidade tanto do *tag array* como do banco de registradores. Através da consulta à informação armazenada, é possível determinar as instruções que necessitam ter seus resultados propagados para essas estruturas. Instruções que sabidamente terão seus registradores destino reescritos por outras instruções que estão sendo despachadas no mesmo ciclo, não necessitarão ter seus resultados propagados para o banco de registradores; apenas para as estações de reserva. Isto diminui o número de portas necessárias nestes dispositivos.

Se admitirmos que a arquitetura tenha pelo menos um comparador por instrução despachada, os valores nos gráficos terão que ser reajustados, pois alguns dos resultados apresentados são menores do que um comparador por instrução. O uso de pelo menos um comparador por instrução pode ser imposição prática, para evitar o uso de um número excessivo de multiplexadores para acesso aos comparadores existentes. Neste caso, o uso de um número grande de multiplexadores poderia representar um gasto maior do que os comparadores economizados.

# Capítulo 6

## Conclusões e Trabalhos Futuros

### 6.1 Conclusões

Os processadores super escalares recentemente lançados no mercado, como o UltraSPARC [85], MIPS R1000 [32], Alpha 21264 [33] e PowerPC604 [39], apresentam largura de despacho de 4 instruções. A evolução natural destes processadores é para arquiteturas com maiores larguras de despacho, em torno de 8 até 16 instruções por ciclo, que denominamos de arquiteturas de amplo despacho.

As técnicas empregadas por esses processadores são inadequadas para o despacho de 8 ou 16 instruções por ciclo. Diversos obstáculos precisam ser superados para que estas larguras de despacho sejam viabilizadas, entre eles destacam-se a predição de desvios e a busca eficiente de múltiplas instruções por ciclo.

Entretanto, mesmo com a superação desses dois obstáculos, ainda é necessário otimizar o estágio de despacho dessas arquiteturas. O despacho eficiente de múltiplas instruções por ciclo é ainda um problema sem solução definitiva proposta na literatura.

Particularmente, o despacho de um grande número de instruções em paralelo apresenta os seguintes obstáculos: o tamanho do banco de registradores é dilatado pela necessidade de um grande número de registradores de renomeação; o número de portas do banco de registradores, das tabelas de renomeação e do *reorder buffer* aumenta consideravelmente; aumenta o *hardware* necessário para a detecção das dependências, que possui complexidade  $O(n^2)$ , onde  $n$  é número de instruções; a difusão dos resultados das instruções executadas pelas unidades funcionais para a

janela de instruções é muito mais complexa.

Por exemplo, uma arquitetura com largura de despacho de 4 instruções requer um banco de registradores com cerca de 8 portas de leitura e 4 de escrita, considerando-se até dois operandos fonte e um resultado por instrução. Ao dobrar a largura de despacho para 8 instruções, 16 portas de leitura e 8 de escrita seriam necessárias. Com o aumento da largura de despacho, também aumenta o número de instruções na janela de registradores aguardando por resultados e, como consequência, aumenta o número de registradores de renomeação necessários.

Para avaliar o impacto do tamanho e do número de portas no tempo de acesso ao banco de registradores, elaboramos um modelo analítico, apresentado em detalhes no Anexo B. Ele mostra que o tempo de acesso aumenta consideravelmente, independentemente do tipo de célula de memória utilizada para armazenar os dados. No exemplo acima, o tempo de acesso ao banco de registradores passaria de 3,3 ns para 3,6 ns (Figura 3.6), ao dobrar o total de portas do banco de registradores, considerando-se uma tecnologia de 0.8  $\mu\text{m}$ . Isto implica em um aumento de cerca de 10% no tempo de ciclo do processador, mesmo utilizando-se a melhor organização de célula de memória que estudamos.

Conforme mostrado no Anexo B, verificamos que dobrando o número de portas de um banco de registradores provoca um impacto no tempo de acesso equivalente ou maior do que dobrar o seu tamanho.

Iniciamos nosso trabalho de investigação com a busca de um método de avaliação conveniente. Primeiramente, o conjunto de programas de avaliação do SPEC95, tanto de inteiros como de ponto flutuante, foi escolhido para ser utilizado em nossos estudos.

Em seguida, determinamos o melhor meio para simular programas com um número tão grande de instruções executadas. Depois de investigar cuidadosamente técnicas de amostragem, como mostrado no anexo A, finalmente optamos pela utilização do simulador *Shade*, que nos ofereceu velocidade de simulação suficiente. Deste modo, todos os programas do SPEC95, sejam inteiros ou de ponto flutuante, foram simulados integralmente, ora com a entrada *reference*, ora com a entrada *training*.

Em seguida, os programas do SPEC95 foram novamente simulados. Desta vez



para determinar o total de instruções executadas e a média dos tamanhos dos blocos básicos executados. Como é sabido, o tamanho médio dos blocos básicos de programas de ponto flutuante é bem maior que o de programas inteiros. Nossos experimentos mostraram que os primeiros apresentam, em média, 25 instruções por cada bloco básico *executado*, enquanto os programas inteiros apenas 5 instruções.

Este pequeno tamanho dos blocos básicos dos programas inteiros oferece uma maior dificuldade para a resolução das dependências de controle que é realizada pelos mecanismos de predição de desvios. Por exemplo, quando do despacho de 16 instruções por ciclo, é necessário prever 3 ou mais desvios por ciclo para os programas inteiros, enquanto que para os programas de ponto flutuante apenas um é suficiente. Além disto, o grande número de blocos básicos distintos requer o armazenamento de uma grande quantidade de informações para que os mecanismos de previsão de desvios sejam eficientes. Só para exemplificar, o programa *gcc* possui cerca de 11.000 blocos básicos distintos que são executados.

Não satisfeitos com os resultados obtidos para os programas inteiros, realizamos uma nova série de testes e observamos os seguintes fatos: nos programas inteiros, o número médio de instruções entre dois desvios que foram tomados é de 10 instruções, isto é, cerca de 10 instruções adjacentes são executadas, sem que ocorra nenhuma interrupção no fluxo de instruções. E mais, no programa *gcc*, por exemplo, dentre os 11.000 blocos básicos, apenas 1060 blocos básicos participam da execução de 90% das instruções do programa.

Isto mostra que, ao contrário do que parecia inicialmente, existem oportunidades para explorar o paralelismo dos programas inteiros, com custo bem menor.

A partir destes experimentos iniciais, especificamos um modelo de arquitetura para investigar mecanismos alternativos de despacho simultâneo de múltiplas instruções em arquiteturas super escalares.

As principais características do nosso modelo de arquitetura são:

- Eliminação de Dependências Falsas

Embora o algoritmo de Tomasulo tenha sido empregado em projetos de pesquisa e modificado em implementações de processadores reais, o nosso modelo apresenta uma característica que não foi explorada nesses projetos. Segundo esta característica, mostrada em detalhes na Seção 3.3, para a resolução efi-

ciente das dependências falsas, o uso de estações de reserva, junto com um controle para priorizar a escrita de dados no *tag array* é suficiente quando múltiplas instruções são despachadas em paralelo.

- Mecanismo de Execução Especulativa

Com o aumento da demanda por registradores de renomeação, o uso de um *future file* em conjunto com um *reorder buffer* para a manutenção de um estado preciso da arquitetura em caso de falha na predição ou interrupção passa a ser vantajoso (Seção 3.6). Este esquema já foi proposto na literatura, mas preterido em favor de outros esquemas, pois o custo de implementação era maior. Entretanto, com o aumento da largura de despacho e do número de registradores necessários para renomeação, o uso de um *future file* reduz o custo e aumenta o desempenho do processador.

O custo é menor porque utiliza-se um número menor de registradores. No nosso modelo o *future file* possui sempre 32 registradores, o mesmo número de registradores do banco arquitetural, independentemente da largura de busca. O desempenho é maior porque as estruturas utilizadas são muito mais simples. Por exemplo, o *reorder buffer* empregado em nosso esquema dispensa a busca associativa utilizada no *reorder buffer* das arquiteturas comerciais atuais.

- Despacho Seletivo de Instruções

Em relação à propagação dos resultados para a janela de instruções, dois esquemas são utilizados tradicionalmente: no esquema de janela centralizada é necessário propagar para o mecanismo de controle da janela o identificador do registrador para onde se destina o resultado produzido. Este mecanismo deve então identificar as instruções que estavam aguardando por ele e, se todos os seus operandos já tiverem sido produzidos, passar o seu estado para “pronto”; no esquema com janela distribuída é preciso propagar o resultado produzido, junto com o seu *tag*, para as estações de reserva. Compete a cada uma das estações de reserva verificar se o resultado produzido é um dos operandos fonte que a instrução armazenada aguarda.

Nas nossas máquinas experimentais equipadas com o despacho seletivo, utilizamos estações de reserva especializadas. Desse modo, as instruções são

transferidas para as estações de acordo com o número de operandos que ainda não foram produzidos. Nossos experimentos mostraram que, no caso dos programas inteiros, entre 50% e 90% das instruções despachadas que são enviadas para estações de reserva ficam aguardando por nenhum ou apenas um dos operandos fonte. No caso dos programas de ponto flutuante, a percentagem varia entre 50% e 66% em quase todos os programas, com exceção do programa *swim* (26%).

Como consequência, o mecanismo de despacho seletivo reduz na mesma proporção o número de comparadores e barramentos para a propagação de resultados.

- *Cache de Dependências*

Para a detecção mais eficiente das dependências diretas, especificamos e simulamos três modelos de *cache* de dependências. Os modelos reduzem a complexidade  $O(n^2)$  do *hardware* que existe nas arquiteturas convencionais.

Nos resultados de nossas simulações, o modelo de *cache* de dependências avançada apresentou os melhores resultados, mesmo para os programas de ponto flutuante. Houve uma redução do número médio de comparadores necessários ao despacho das instruções de um total de 197 para apenas 15, no caso dos programas inteiros, e de 211 para apenas 10, no caso de programas de ponto flutuante, ambos com uma largura de despacho de 16 instruções. Houve uma redução do número médio de comparadores necessários ao despacho das instruções de um total de 46 para apenas 4, no caso dos programas inteiros, e de 50 para apenas 2, no caso de programas de ponto flutuante, quando a largura de despacho for de 8 instruções.

Mesmo para a *cache* de dependências inteligente, que possui menor complexidade de implementação, o número médio de comparadores necessários foi reduzido para de 197 para 63 e de 211 para 63 comparadores, respectivamente para os programas inteiros e de ponto flutuante, com largura de despacho de 16 instruções. O número médio de comparadores necessários foi reduzido de 46 para 10 comparadores para os programas inteiros e de 50 comparadores para 12 nos de ponto flutuante, com largura de despacho de 8 instruções.

Na *cache* simples, de menor complexidade de implementação, a redução é de 197 para 90 comparadores no caso dos programas inteiros e de 211 para 126 registradores com os programas de ponto flutuante, também para uma largura de despacho de 16 instruções. Para uma largura de 8 instruções, a redução é de 46 para 21 comparadores no caso dos programas inteiros e para 50 para 30 comparadores com os programas de ponto flutuante.

Estes resultados são ideais, pois estamos supondo um número ilimitado de recursos. Implementações reais deste mecanismo devem apresentar um desempenho mais modesto do que os obtidos nos nossos experimentos. Os custos associados a estas implementações devem ser maiores, pois apenas contabilizamos o custo associado ao armazenamento dos dados na *cache* de dependências. Acreditamos porém que o potencial de nossos mecanismos tenha sido demonstrado plenamente, pela significativa redução no número de comparadores e barramentos.

Na realidade, os mecanismos que manipulam a *cache* de dependência podem (e devem) ser associados a outros esquemas de armazenamento, como por exemplo uma *cache* de blocos básicos ou com uma *trace cache*. Neste caso os custos são compartilhados pelos benefícios oferecidos por estes dispositivos, como diminuição da largura de banda necessária para a busca de instruções.

Não menos importante, as estruturas de dados apresentadas permitem identificar os resultados gerados que precisam ser enviados para o registrador arquitetural e/ou *future file*. Os resultados que serão usados apenas para instruções do mesmo bloco básico, serão transmitidos apenas para as estações de reserva que estiverem aguardando por eles. Isto permite reduzir o número de portas do banco de registradores e do *tag array*, resultando em dispositivos mais simples e com menor tempo de acesso.

Finalmente, procuramos nesta tese ampliar o entendimento do despacho de múltiplas instruções e oferecer soluções eficientes para a exploração de paralelismo no nível de instrução nas arquiteturas super escalares.

## 6.2 Trabalhos Futuros

Como trabalhos futuros pretendemos investigar o uso de uma *cache* de blocos básicos, acoplado à *cache* de dependências. Tamanhos realistas de *cache* serão então analisados e os resultados obtidos permitirão dimensionar com precisão os gastos e benefícios com a implementação dessas estruturas.

Além deste trabalho, continuamos nossa investigação de métodos que permitam diminuir o tamanho do banco de registradores. Apesar de não termos simulado, apresentamos no anexo C uma alternativa para diminuir o tempo de alocação dos registradores de renomeação em uma arquitetura super escalar.

A implementação usual de especulação dinâmica, interrupção precisa e eliminação de dependências, utiliza tabelas de renomeação de registradores, janela de instruções centralizada e *reorder buffers* [34, 86, 42]. A renomeação e alocação dos registradores físicos é feita no estágio de decodificação e/ou busca das instruções, ou mesmo em um estágio dedicado para a renomeação, entre a decodificação e o despacho.

Adaptações do algoritmo de Tomasulo para especulação dinâmica e interrupção precisa, em uso nas arquiteturas projetadas recentemente [39, 32, 33], também fazem uso de tabelas de renomeação e de um *reorder buffer* com busca associativa para fornecer os resultados requeridos pelas instruções que estão sendo despachadas, mas que ainda não foram atualizados nos registradores arquiteturais. A renomeação é feita no estágio de despacho nesses casos.

Contudo, o número de registradores físicos tende a aumentar consideravelmente, quando aumenta a largura de despacho e o tamanho da janela de instruções [35, 36].

Nesta tese apresentamos um mecanismo que utiliza o *future file*, que é eficiente quando o número de registradores de renomeação necessários é equivalente ou maior que o número de registradores arquiteturais. Em alguns casos, contudo, a utilização de esquemas alternativos que evitem a demanda por um número tão grande de registradores de renomeação se torna interessante.

Trabalhos recentes apresentam soluções que usam o particionamento da janela de instruções e do banco de registradores [63]; que fazem a reutilização de registradores físicos [87]; ou que atrasam ao máximo a alocação dos registradores para estágios posteriores do *pipeline* [73, 72], neste último caso à custa da re-execução de algumas

instruções.

No anexo C, apresentamos os detalhes de uma alternativa, baseada em arquiteturas com janela de instruções distribuídas, que permite que a alocação dos registradores físicos seja feita apenas no estágio de execução, sem contudo, necessitar de esquemas de re-execução de instrução ou fazer uso da reutilização de registradores.

## Capítulo 7

### Referências Bibliográficas

# Bibliografia

- [1] Smith, M. D. , Johnson, M. , Horowitz, M. A. , “Limits on multiple instruction issue”, In: *Proceedings of 3rd International Conference on Architectural Support for Prog. Lang. and Operating Systems*, pp. 290-302, ACM, April 1989.
- [2] Austin, T. M. , Sohi, G. S. , “Dynamic dependency analysis of ordinary programs”, In: *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 342-351, ACM, May 1992.
- [3] Wall, D. W. , “Limits of instruction level parallelism”, Tech. Rep. WRL-TR-93/6, Western Research Laboratory - Digital, November 1993.
- [4] Mudge, T. N. , Chen, I.-C. K. , Coffey, J. T. , “Limits to branch prediction”, Tech. Rep. CSE-TR-282-96, EECS - University of Michigan, 1996.
- [5] Nair, R. , Hopkins, M. E. , “Exploiting instruction level parallelism in processors by caching scheduled groups”, In: *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 13-25, ACM, 1997.
- [6] Postiff, M. , Tyson, G. , Mudge, T. , “Performance limits of trace caches”, Tech. Rep. CSE-TR-373-98, Advanced Computer Architecture Laboratory - University of Michigan, September 1998.
- [7] Postiff, M. , Greene, D. , Tyson, G. , Mudge, T. , “The limits of instruction level parallelism in SPEC95 applications”, In: *Proceedings of the 3rd Workshop on Interaction Between Compilers and Computer Architecture - ASPLOS VIII*, pp. 31-34, ACM, 1998.



- [8] Martel, I. , Ortega, D. , Ayguadé, E. , Valero, M. , “Increasing effective IPC by exploiting distant paralellism”, In: *Proceedings of the 1999 International Conference on Supercomputing*, pp. 348-255, ACM, 1999.
- [9] Bennett, J. E. , Flynn, M. J. , “Performance factors for superscalar processors”, Tech. Rep. CSL-TR-95-661, Computer Systems Laboratory - Stanford University, February 1995.
- [10] Yeh, T.-Y. , Patt, Y. N. , “Alternative implementations of two-level adaptive branch prediction”, In: *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 124-134, ACM, May 1992.
- [11] Yeh, T.-Y. , Marr, D. T. , Patt, Y. N. , “Increasing the instruction fetch rate via multiple branch prediction and a branch address cache”, In: *Proceedings of the 1993 International Conference on Supercomputing*, pp. 67-76, ACM, July 1993.
- [12] Calder, B. , Grunwald, D. , “Fast and accurate instruction fetch and branch prediction”, In: *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 2-11, ACM, April 1994.
- [13] Chang, P.-Y. , Hao, E. , Patt, Y. N. , “Alternative implementations of hybrid branch predictors”, In: *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 252-257, IEEE, December 1995.
- [14] Young, C. , Gloy, N. , Smith, M. D. , “A comparative analysis of schemes for correlated branch prediction”, In: *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 276-286, ACM, June 1995.
- [15] Driesen, K. , Hólzle, U. , “Limits of indirect branch prediction”, Tech. Rep. TRCS97-10, University of California at Santa Barbara, June 1997.
- [16] Stark, J. , Evers, M. , Patt, Y. N. , “Variable length path branch prediction”, In: *Proceedings of 8th International Conference on Architectural Support for Prog. Lang. and Operating Systems*, pp. 170-179, ACM, 1998.

- [17] Lipasti, M. H. , Wilkerson, C. B. , Shen, J. P. , “Value locality and load value prediction”, In: *Proceedings of the 7th International Conference on Architectural Support for Progr. Lang. and Operating Systems*, pp. 138-147, ACM, September 1996.
- [18] Lipasti, M. H. , Shen, J. P. , “Exceeding the dataflow limit via value prediction”, In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226-237, IEEE, December 1996.
- [19] Sodani, A. , Sohi, G. S. , “Dynamic instruction reuse”, In: *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205, ACM, July 1997.
- [20] Gabbay, F. , Medelson, A. , “Using value prediction to increase the power of speculative execution hardware”, *Transaction on Computer Systems(TOCS)*, v. 16, n. 3, pp. 234-270, August 1998.
- [21] Sohi, G. S. , Vajapeyam, S. , “Instruction issue logic for high-performance, interruptable pipelined processors”, In: *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 27-34, ACM, 1987.
- [22] Wen, C.-P. , “Improving instruction supply efficiency in superscalar architectures using instruction trace buffers”, In: *Proceedings of the 1992 Symposium on Applied Computing (vol I)*, pp. 28-36, ACM/SIGAPP, 1992.
- [23] Conte, T. M. , Menezes, K. N. , Mills, P. M. , Patel, B. A. , “Optimization of instruction fetch mechanisms for high issue rates”, In: *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 333-344, ACM, 1995.
- [24] Lee, D. , Baer, J.-L. , Calder, B. , Grunwald, D. , “Instruction cache fetch policies for speculative execution”, In: *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 357-367, ACM, 1995.
- [25] Uhlig, R. , Nagle, D. , Mudge, T. *et al.*, “Instruction fetch: Coping with code bloat”, In: *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 345-356, ACM, June 1995.

- [26] Rotenberg, E. , Bennett, S. , Smith, J. E. , "Trace cache: A low latency approach to high bandwidth instruction fetching", In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-35, IEEE, December 1996.
- [27] Yung, R. , "Design of the UltraSPARC instruction fetch unit", Tech. Rep. SMLI TR-96-59, Sun Microsystems Laboratories, December 1996.
- [28] Chen, I.-C. K. , Lee, C.-C. , Mudge, T. , "Instruction prefetching using branch prediction information", In: *Proceedings of the 1997 International Conference on Computer Design*, pp. 593-601, 1997.
- [29] Chen, I.-C. K. , *Enhancing the Instruction Fetch Mechanism using Data Compression*. PhD thesis, University of Michigan, 1997.
- [30] Michaud, P. , Seznec, A. , Jourdan, S. , "Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors", Tech. Rep. 1227, IRISA, February 1999.
- [31] Butler, M. , Patt, Y. N. , "An investigation of the performance of various dynamic scheduling techniques", In: *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 1-9, ACM, 1992.
- [32] Yeager, K. C. , "MIPS R10000 superscalar microprocessor", *IEEE Micro*, pp. 28-40, April 1996.
- [33] Kessler, R. E. , "The Alpha 21264 microprocessor", *IEEE Micro*, v. 19, n. 2, pp. 24-36, March/April 1999.
- [34] Johnson, M. , *Superscalar Microprocessor Design*. Englewood Cliffs, New Jersey: Prentice-Hall, 1991.
- [35] Silva, G. P. , Chaves, E. M. , Goulart, V. M. , Alves, V. C. , "An analytical area, access and cycle time model for on-chip multiported memories", *Journal of Solid-State Devices and Circuits*, v. 6, n. 1, pp. 17-23, February 1998.
- [36] Farkas, K. I. , Jouppi, N. P. , Chow, P. , "Register file design considerations in dynamically scheduled processors", Tech. Rep. WRL-TR-95/10, Western Research Laboratory - Digital, November 1995.

- [37] Palachala, S. , Jouppi, N. P. , Smith, J. E. , “Quantifying the complexity of superscalar processors”, Tech. Rep. CS-TR-1996-1328, CS/ECE - University of Wisconsin-Madison, November 1996.
- [38] Tomasulo, R. M. , “An efficient algorithm for exploiting multiple arithmetic units”, *IBM Journal of Research and Development*, v. 11, n. 1, pp. 25-33, January 1967.
- [39] IBM, Motorola, *PowerPC 604 - RISC Microprocessor User's Manual*, 1994.
- [40] Intel, *P6 Family of Processors - Hardware Developer's Manual*. Intel, September 1998.
- [41] Williams, T. , Patkar, N. , Shen, G. , “SPARC64: A 64-b 64-active-instruction out-of-order-execution MCM processor”, *IEEE Journal of Solid-State Circuits*, v. 30, n. 11, pp. 1215-1226, November 1995.
- [42] Smith, J. E. , Pleszkun, A. R. , “Implementation of precise interrupts in pipelined processors”, In: *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 291-299, ACM, 1985.
- [43] Yeh, T.-Y. , Patt, Y. N. , “Two-level adaptative branch prediction”, In: *Proceedings of the 24th International Symposium on Microarchitecture*, pp. 51-61, IEEE, November 1991.
- [44] Evers, M. , Patel, S. J. , Campbell, R. S. , Patt, Y. N. , “An analysis of correlation and predictability: What makes two-level predictors work”, In: *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 52-61, ACM, June 1998.
- [45] Smith, J. E. , “A study of branch prediction strategies”, In: *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, ACM, May 1981.
- [46] Young, C. , Smith, M. D. , “Improving the accuracy of static branch prediction using branch correlation”, In: *Proceedings of the 6th International Conference on Architectural Support for Prog. Lang. and Operating Systems*, pp. 232-241, ACM, October 1994.

- [47] Gloy, N. , Smith, M. D. , Young, C. , “Performance issues in correlated branch prediction schemes”, Tech. Rep. TR-23-95, Harvard University, 1995.
- [48] Chen, I.-C. K. , Lee, C.-C. , Postiff, M. A. , Mudge, T. , “Tagless two-level branch prediction schemes”, Tech. Rep. CSE-TR-306-96, University of Michigan, September 1996.
- [49] McFarling, S. , “Combining branch predictors”, Tech. Rep. WRL-TN-36, Western Research Laboratory - Digital, June 1993.
- [50] Dutta, S. , Franklin, M. , “Control flow prediction with tree like subgraphs for superscalar processors”, In: *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 258-263, IEEE, December 1995.
- [51] Wallace, S. , Bagherzadeh, N. , “Instruction fetch mechanisms for superscalar microprocessors”, In: *Proceedings of the Europar96*, pp. 747-756, 1996.
- [52] Wallace, S. , Bagherzadeh, N. , “Multiple branch and block prediction”, In: *Proceedings of the Third International Symposium on High Performance Computer Architecture*, pp. 94-105, IEEE, February 1997.
- [53] Sez nec, A. , Jourdan, S. , Sainrat, P. , Michaud, P. , “Multiple block-ahead branch predictors”, In: *Proceedings of the 7th International Conference on Architectural Support for Prog. Lang. and Operating Systems*, pp. 116-127, ACM, October 1997.
- [54] Patel, S. J. , Friendly, D. H. , Patt, Y. N. , “Critical issues regarding the trace cache fetch mechanism”, Tech. Rep. CSE-TR-335-97, EECS - University of Michigan, 1997.
- [55] Nair, R. , “Dynamic path-based branch correlation”, In: *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 15-23, IEEE, December 1995.
- [56] Jacobson, Q. , Bennett, S. , Sharma, N. , Smith, J. E. , “Control flow speculation in multiscalar processors”, In: *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pp. 218-229, IEEE, February 1997.

- [57] Jacobson, Q. , Rotenberg, E. , Smith, J. E. , “Path-based next trace prediction”, In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 14-23, IEEE, December 1997.
- [58] Patel, S. J. , Evers, M. , Patt, Y. N. , “Improving trace cache effectiveness with branch promotion and trace packing”, In: *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 262-271, ACM, 1998.
- [59] Stark, J. , Racunas, P. , Patt, Y. N. , “Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order”, In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 34-43, IEEE, 1997.
- [60] Sprangle, E. , Patt, Y. , “Facilitating superscalar performance via a combined static/dynamic register renaming scheme”, In: *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 143-147, IEEE, November 1994.
- [61] Melvin, S. , Patt, Y. , “Enhancing instruction scheduling with a block structured ISA”, *International Journal of Parallel Programming*, v. 23, n. 3, pp. 221-243, 1995.
- [62] Hao, E. , Chang, P.-Y. , Evers, M. , Patt, Y. N. , “Increasing the instruction fetch rate via block-structured instruction set architectures”, In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 191-200, IEEE, December 1996.
- [63] Vajapeyam, S. , Mitra, T. , “Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences”, In: *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 1-12, ACM, May 1997.
- [64] Fernandes, E. S. T. , *Paralelismo a Nível de Instrução e o Custo dos Desvios*. 11<sup>a</sup> Escola de Computação - DCC/IM, COPPE/Sistemas, NCE/UFRJ, Julho 1998.
- [65] Kaeli, D. R. , Emma, P. G. , “Branch history table predictions of moving target branches due to subroutine returns”, In: *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 34-42, ACM, 1991.

- [66] Friendly, D. H. , Patel, S. J. , Patt, Y. N. , “Alternative fetch and issue policies for the trace cache fetch mechanism”, In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 24-33, IEEE, December 1997.
- [67] Melvin, S. , Patt, Y. , “Exploiting fine-grained parallelism through a combination of hardware and software techniques”, In: *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 287-297, ACM, 1991.
- [68] Hwu, W. W. , Mahlke, S. , Chen, W. *et al.*, “The superblock: An effective technique for VLIW and superscalar compilation”, *Journal of Supercomputing*, v. 7, n. 9, pp. 229-248, 1993.
- [69] Rotenberg, E. , Bennett, S. , Smith, J. E. , “Trace cache: A low latency approach to high bandwidth instruction fetching”, Tech. Rep. CS-TR-96-1310, Computer Science - University of Wisconsin-Madison, April 1996.
- [70] Friendly, D. H. , Patel, S. J. , Patt, Y. N. , “Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors”, In: *Proceedings of 31st International Symposium on Microarchitecture*, pp. 173-181, IEEE, December 1998.
- [71] Franklin, M. , Sohi, G. S. , “Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors”, In: *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 34-45, IEEE, December 1992.
- [72] González, A. , González, J. , Valero, M. , “Virtual-physical registers”, In: *Proceedings of the 4th International Symposium on High Performance Computer Architecture*, pp. 175-184, IEEE, February 1998.
- [73] Monreal, T. , González, A. , Valero, M. *et al.*, “A novel renaming scheme to exploit value temporal locality through physical register reuse and unification”, In: *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 216-225, IEEE, December 1998.

- [74] Fernandes, E. S. T. , Wolfe, A. , Silva, G. P. , “Towards BBM - a basic block machine”, Tech. Rep. 526/00, COPPE/Sistemas, Federal University of Rio de Janeiro, January 2000.
- [75] Fernandes, E. S. T. , Silva, G. P. , “BBM - um processador de blocos básicos”, In: *Anais do I Workshop em Sistemas Computacionais de Alto Desempenho*, Outubro 2000.
- [76] Paterson, D. A. , Hennessy, J. L. , *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
- [77] Huang, J. , Lilja, D. J. , “Exploiting basic block value locality with block reuse”, In: *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, pp. 106-114, IEEE, January 1999.
- [78] Cmelick, R. F. , Keppel, D. , “Shade: A fast instruction-set simulator for executing profiling”, Tech. Rep. SMLI TR-93-12, Sun Microsystems Laboratories, 1993. Also published as Tech. Report CSE-TR 93-06-06, University of Washington.
- [79] Silva, G. P. , Fernandes, E. S. T. , “Técnicas para a avaliação de arquiteturas superescalares”, In: *Proceedings of the IX Brazilian Symposium on Computer Architectures*, (Campos do Jordão), pp. 269-283, SBAC, October 1997.
- [80] Dubey, P. K. , Nair, R. , “Profile-driven sampled trace generation”, Tech. Rep. RC-20041, IBM Research Division, April 1995.
- [81] Sun Microsystems, Mountain View, CA, *Introduction to Shade*, June 1997.
- [82] Cvetanovic, Z. , Bhandarkar, D. , “Characterization of Alpha AXP performance using TP and SPEC workloads”, In: *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 60-70, IEEE, 1994.
- [83] Ball, T. , Larus, J. R. , “Efficient path profiling”, In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 46-57, IEEE, December 1996.
- [84] Reily, J. , “SPEC describes SPEC95 products and benchmarks”, *SPEC Newsletter*, September 1995. <http://www.spec.org/osg/news/articles/news9509/cpu95descr.html>.



- [85] Yung, R. , *Evaluation of a Commercial Microprocessor*. PhD thesis, University of California, Berkeley, June 1998. Also SMLI TR-98-65 Sun Microsystems Laboratories.
- [86] Moudgill, M. , Pingali, K. , Vassiliadis, S. , “Register renaming and dynamic speculation: An alternative approach”, In: *Proceedings of the 26th International Symposium on Microarchitecture*, pp. 202-213, IEEE, 1993.
- [87] Jourdan, S. , Ronen, R. , Bekerman, M. *et al.*, “A novel renaming scheme to exploit value temporal locality through physical register reuse and unification”, In: *Proceedings of the 31st International Symposium on Microarchitecture*, pp. 216-225, IEEE, December 1998.

## **Apêndice A**

# **Técnicas para Avaliação do Desempenho de Arquiteturas Super Escalares**

# Técnicas para Avaliação do Desempenho de Arquiteturas Super Escalares

Gabriel Pereira da Silva<sup>®</sup>; Edil S. T. Fernandes\*  
(<sup>®</sup>NCE e COPPE-Sistemas) (\*COPPE-Sistemas)

Universidade Federal do Rio de Janeiro

Cx. Postal 68511 - CEP 21945-970

Rio de Janeiro - RJ

e-mail: gabriel@cos.ufrj.br; edil@cos.ufrj.br

## Resumo

Durante o projeto de um processador é necessário realizar inúmeras simulações para determinar as melhores opções que serão incorporadas na sua arquitetura. *Trace-driven simulation* é uma das técnicas mais utilizadas. O *trace* obtido é processado pelo simulador do modelo de arquitetura em teste e a partir daí são extraídas estatísticas de desempenho do modelo. Para cada modificação na arquitetura investigada, precisamos simular novamente o mesmo conjunto de programas de teste. Neste artigo é apresentado um método alternativo de avaliação que usa diversas amostras, igualmente espaçadas ao longo da execução do programa, para formar um *trace* reduzido, porém representativo, da execução completa de cada programa de teste. Este *trace* amostrado é utilizado como entrada pelo simulador de uma arquitetura super escalar, permitindo que as modificações sejam avaliadas em um tempo muito mais reduzido. Experimentos com programas de teste mostraram que com apenas 1% das instruções é possível reproduzir o comportamento da arquitetura super escalar com o *trace* completo.

## Abstract

Trace-driven simulation is a powerful technique to help designers during the specification of new machines. The technique allows to reproduce the machine behavior even before its actual implementation. For this reason, it plays an important role to assess the performance of new superscalar processors. In order to achieve a higher degree of parallel activities, the architecture of these very sophisticated processors must be very well balanced, and the evaluation of the performance impact caused by each modification in the basic architecture model requires the simulation of the overall set of test programs: a very time consuming task because each benchmark program consists of billions of instructions demanding a very large simulation time. In this paper, we present an alternative evaluation method that uses several samples, evenly distributed in time, to compose a reduced but representative trace from the complete benchmark execution. This sampled trace is used as input to the architecture model, and it allows a faster assessment of the effect caused by each modification. Experiments with test programs have shown that only with 1% of the instructions is possible to reproduce the behavior of the superscalar architecture with the complete trace.

## A.1 Introdução

No projeto da arquitetura de um processador é imprescindível a realização de simulações para determinar as melhores opções a serem utilizadas na sua arquitetura. Uma das técnicas mais utilizadas é a de *trace-driven simulation*. Neste método, é gerado um *trace* dinâmico dos programas de avaliação, que contém uma seqüência dinâmica dos endereços e códigos das instruções, e dos endereços dos dados que ocorrem durante a execução do programa. Este *trace* é aplicado ao modelo de arquitetura em teste, e a partir daí são levantadas estatísticas de desempenho do modelo. É necessária uma nova execução de todo o conjunto de programas de avaliação para cada modificação na arquitetura a ser avaliada.

A necessidade de melhor avaliar o desempenho dos microprocessadores, levou a elaboração de sofisticadas suítes de programas (p.ex. SPEC92), de forma a representar uma aplicação típica de usuário. Estas suítes executam alguns bilhões de instruções, representando um problema para o projetista, quando avaliando as opções de projeto de sua arquitetura. A execução completa da suíte SPEC92 requer mais de 100 bilhões de instruções para cada modelo de arquitetura. Com velocidades de simulação na ordem de dezenas de milhares de instruções por segundo, a simulação completa da suíte de avaliação levaria vários dias.

Além dos tempos envolvidos, uma outra questão importante é a quantidade de espaço em disco gasta para armazenar os *traces*. Um programa com 1 bilhão de instruções, considerando-se apenas o espaço gasto para armazenar as referências de busca das instruções, precisaria de 4 Gbytes de espaço de armazenamento. Algumas técnicas para redução do espaço gasto podem ser aplicadas [PLE94], mas o espaço gasto ainda é considerável.

Para tornar esta tarefa viável, propomos neste trabalho a utilização de técnicas de amostragem para obtenção dos *traces*. O método de amostragem proposto neste trabalho é o de utilizarmos diversas amostras, igualmente espaçadas ao longo da execução do programa, para formar um conjunto representativo do programa de avaliação. Esta técnica foi utilizada na avaliação de diversos processadores super escalares comerciais [SHI94, LAU93], com variações que serão detalhadas ao longo deste trabalho. Apresentaremos a seguir os resultados deste trabalho, realizado para otimizar os tempos de execução e espaço de armazenamento gastos na realização de

estudos de arquitetura de processadores no programa de Engenharia de Sistemas da COPPE/UFRJ.

## A.2 Trabalhos Anteriores

O uso de técnicas de amostragem se aplica tanto para a avaliação do comportamento de *caches* [LAH88, KES91, LIU93], como o de modelos de arquitetura de processadores [LAU93,SHI94].

Esta técnica foi utilizada também no desenvolvimento de um monitor de desempenho [MAR93]. Nesse desenvolvimento, o total de amostras utilizado correspondeu a 10% do total de instruções, com cerca de 20 amostras de 0.5 MB cada. Quando as *caches* simuladas possuíam tamanhos de 16 KB e 128KB, o erro absoluto na taxa de falhas da *cache* não excedia a 0.3%. Para tamanhos maiores de *cache* (aprox. 1 MB), o uso de amostras maiores (4 a 8 milhões de referências cada) permite melhorar a acurácia, mantendo-se a mesma taxa de amostragem (10%).

Outros trabalhos [LAH88] indicam que 35 amostras são suficientes para caracterizar bem (erro percentual menor que 5,5%) a taxa de falhas para aplicações LISP, quando a taxa de amostragem se situa entre 3 e 10%. Em [MAR93] encontramos curvas mostrando que, mantendo-se o mesmo percentual de 10%, a partir de 10 amostras o erro em relação a taxa de falhas real é menor que 5% para a maioria dos programas. A exceção ocorre nos programas onde a taxa de erro absoluta é muito pequena (p.ex., espresso).

Técnicas de amostragem foram também utilizadas no desenvolvimento de processadores comerciais. No PowerPC [SHI94], por exemplo, as técnicas utilizadas são bastante simples, consistindo de 200 amostras de 5000 instruções contínuas, distribuídas uniformemente ao longo do tempo de execução, totalizando 1.000.000 de instruções, independente do tamanho e do tipo do programa em estudo. Este procedimento não é recomendável, já que o percentual de instruções amostrado variou com cada programa. Para alguns programas de teste o tamanho do *trace* amostrado foi pequeno e o tamanho de cada amostra comprometeu a acurácia das taxas de falha das *caches*, provocando alguns desvios em relação ao *trace* completo.

Nesse trabalho [SHI94], embora o erro da média geométrica do número de Ins-

truções executadas Por Ciclo (IPC) estivesse dentro de 2%, a margem de erro para os *traces*, individualmente, chegou a 13% como no programa *sc*. Em outro tipo de programa utilizado nesta mesma avaliação (de nome TPC), a influência do estado da *cache* no início de cada amostra foi tão grande, que o IPC amostrado se afastou de maneira inaceitável do IPC obtido com o *trace* completo. O uso de amostras maiores, aumentando o percentual de amostragem, poderia reduzir estas distorções.

Em outro trabalho analisado [LAU93] foram realizadas amostragens menores que 0.5% do *trace* completo, significando tempos de simulação 200 vezes menores. Nesse caso, o número inicial de amostras foi de apenas 15, cada uma delas com 250.000 instruções uniformemente distribuídas ao longo do *trace* completo. Caso as amostras tomadas não fossem representativas do comportamento do programa, novas amostras eram então agregadas até obter um conjunto representativo de amostras do programa. Nas amostras não foram incluídas instruções do espaço do sistema (chamadas ao sistema, *traps*, interrupções, etc.).

Existem várias medidas para determinar quando o conjunto de amostras é representativo. Em [LAU93] a verificação foi dividida em duas etapas: verificações rápidas para determinar se o conjunto tomado é claramente insuficiente, e outras mais elaborados para uma verificação final no conjunto de amostras. As verificações mais rápidas usam a frequência das instruções e das funções e estatísticas de *cache*, além do IPC, para determinar quando o conjunto de amostras é suficiente. A verificação mais elaborada compara o desempenho de um conjunto de micro-arquiteturas obtidos pelo *trace* amostrado com o obtido pelo completo. Diferenças sempre menores que 0,5% na frequência de execução das instruções foram obtidas, por exemplo, para o programa *gcc*. O estado inicial da memória *cache* foi reproduzido no início de cada amostra, tornando a diferença absoluta da taxa de falhas sempre menor que 2% entre o *trace* amostrado e o completo. A diferença entre o IPC do SPECint92 para os dois *traces* não foi maior que 3% para os diferentes tipos de arquiteturas simuladas.



## A.3 Metodologia

Nosso objetivo principal consiste em obter *traces* para avaliar com confiabilidade e mais rapidamente diferentes configurações super escalares. Neste artigo apresentamos quando e como devem ser obtidos estes *traces* para que os objetivos sejam atingidos.

Utilizamos em nosso ambiente de teste um simulador super escalar parametrizável com conjunto de instruções compatível com o i860 [INT91] e que utiliza o algoritmo de Tomasulo [TOM67] para resolução dinâmica de dependências. O simulador, denominado *tomasulo* [HAO93], reproduz o comportamento de diversas configurações derivadas do modelo básico ao modificarmos um dos parâmetros arquiteturais (número de unidades funcionais, estações de reserva, número de instruções despachadas simultaneamente, tamanho da *cache* de dados e instruções, tamanho da *cache* de endereços de desvio (BTB) [LEE84], entre outros). Como resultado o simulador fornece o número de ciclos gastos na execução do programa, o *speed-up* em relação a arquiteturas convencionais, taxas de acerto nas *caches* e no BTB, taxas de ocupação das unidades funcionais e estações de reserva, entre outros.

A questão levantada é se poderíamos realizar comparações válidas entre diferentes configurações de arquitetura com o uso da técnica *trace* amostrado.

Para a geração dos *traces*, o simulador *sim860a* [FER93], desenvolvido para o ambiente DOS, foi adaptado para um ambiente UNIX, e rotinas emulando chamadas ao sistema operacional foram acrescentadas. Outra modificação introduzida consistiu na introdução de parâmetros indicando o número e a taxa de amostragem.

O simulador *sim860a* não anota as instruções executadas durante as chamadas ao sistema operacional, já que as mesmas são emuladas pelo processador/sistema operacional hospedeiro. Avaliando a arquitetura Alpha [CVE94] com os programas SPEC92, verificou-se que o percentual de tempo gasto em chamadas ao sistema operacional é menor que 3% para a maior parte dos programas. A exceção é o programa *compress*, que por utilizar *pipes*, gasta perto de 15% do tempo em chamadas ao sistema operacional, não sendo por isto utilizado em nossa análise.

A saída do simulador *sim860a* pode ser redirecionada para um programa compactador e, para cada 1.000.000 de instruções simuladas, obtém-se em média um arquivo com 300 Kbytes, ou seja, uma redução de espaço de de 1:200. Os arquivos

compactados são enviados para um descompactador e daí, via *pipe*, para o simulador *tomasulo*. Assim podemos manter os *traces* sempre compactados, com grande economia de espaço em disco.

Podemos também executar simulações em paralelo em diversas máquinas, com grande economia de tempo. No estudo realizado, simulamos programas com até 100 milhões de instruções.

Considerando-se que o modelo de arquitetura contém somente uma *cache* (para dados e instruções) com 8 Kbytes, não utilizamos nenhuma forma de previsão do estado das *caches*, no início de cada amostra.

Também desenvolvemos um programa para verificar a distribuição das instruções, individualmente e por tipo de unidade funcional, em cada um dos *traces*, para verificar a sua representatividade.

## A.4 A Escolha da Amostras

Analizamos quatro programas: os inteiros *espresso* e *xlisp*; *whetstone* e *tomcatv* de ponto flutuante. Analizamos *traces* incluindo 1% e 100% das instruções de cada programa. O número de amostras variou entre uma dezena e algumas centenas de amostras, dependendo das características do programa de teste. A Tabela 1 lista o número de amostras para cada caso.

Programa	Total de Instruções	Tamanho das Amostras	Número de Amostras
Espresso	1.230.000	1.000	12
Whetstone	2.190.000	1.000	22
Xlisp	15.500.000	1.000	160
Tomcatv	75.000.000	100.000	75

Tabela A.1: Programas de Teste

As amostras são formadas pelas instruções iniciais de cada intervalo de amostragem e contêm 100.000 ou 1.000.000 de instruções, de acordo com o tamanho do programa, mas sempre no percentual de 1% do total de instruções. As entradas de dados do SPEC92 foram alteradas, para permitir tempos de execução e tamanhos de *traces* viáveis.

Para cada um dos programas, produzimos gráficos mostrando a frequência de execução de cada instrução, tanto no *trace* amostrado, como no *trace* completo.

Como queremos validar o uso da técnica de amostragem para estudos comparativos entre diferentes arquiteturas, os dois tipos de *trace* foram submetidos ao simulador *tomasulo* com as configurações de arquitetura A e B, descritas na Tabela 2.

	MÁQUINA A					MÁQUINA B				
	Soma	Mult	Graf	Memo	Core	Soma	Mult	Graf	Memo	Core
FU	2	2	1	1	1	2	2	1	1	3
RS	4	4	2	2	2	10	10	5	5	15

Tabela A.2: Configurações de Arquitetura

Os resultados da simulação de cada arquitetura com os dois *traces* são sumarizados nas tabelas a seguir. Os dados mais significativos são o total de ciclos gastos na execução, taxa de acertos na *cache* de endereços de desvio (BTB), *speed-up* em relação a arquiteturas convencionais, e a taxa de ocupação das unidades funcionais, especificadas por tipo e número. O *speed-up* apresentado refere-se ao IPC da arquitetura super escalar dividido pelo IPC de uma arquitetura seqüencial.

## A.5 Análise dos Resultados

### A.5.1 Espresso

	MÁQUINA A	MÁQUINA B
Tempo total:	41.706 ciclos	37.886 ciclos
<i>Speed-Up</i> :	2,00	2,20
Hit no BTB :	77,21 %	77.21 %

Tabela A.3: Estatísticas Espresso 1%

O programa espresso é um dos menores programas simulados. Ele executa 1.230.000 instruções e o *trace* amostrado possui 12 amostras de 1.000 instruções. As Tabelas 3 e 4 mostram que é um programa com elevadas taxas de acerto no (BTB), e as variações no *speed-up* indicam que é sensível a mudanças nas taxas de

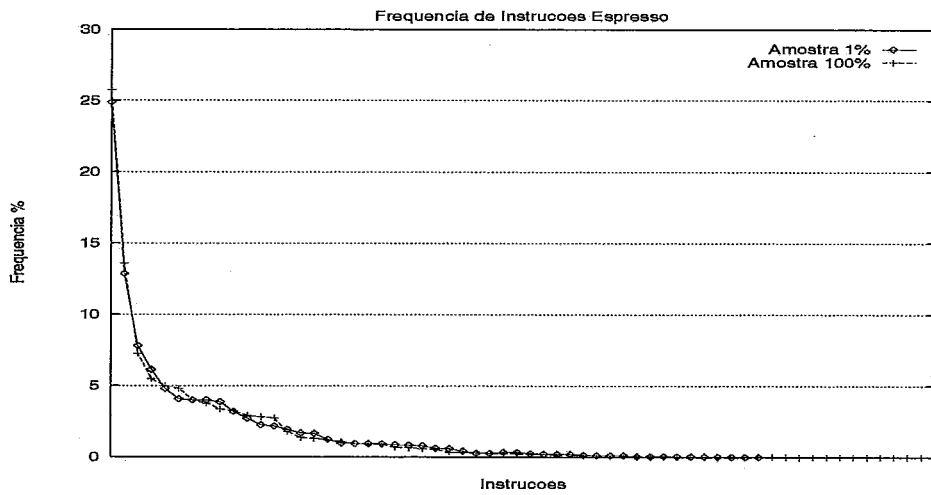


Figura A.1: Frequência das Instruções – Espresso

	MÁQUINA A	MÁQUINA B
Tempo total:	3.745.460 ciclos	3.365.880 ciclos
<i>Speed-Up</i> :	2,10	2,34
Hit no BTB :	92.08%	92.08%

Tabela A.4: Estatísticas Espresso 100%

amostragem. O pequeno tamanho das amostras afeta diretamente a taxa de acerto nas duas *caches* (instruções/dados e BTB), aumentando a taxa de falhas no BTB e reduzindo o *speed-up* absoluto. Mesmo assim, as diferenças relativas entre as arquiteturas A e B são mantidas (10% no *trace* amostrado e 11,4% no completo), o que ainda permite obter dados comparativos válidos.

Nas Tabelas 5 e 6, verificamos que a diferença entre a taxa de distribuição das instruções por unidade funcional não é significativa (não é maior que 1,5%) , em valores absolutos.

A Figura 1 mostra a frequência das instruções dos *traces* amostrado e completo provenientes do programa Espresso. Conforme podemos ver, ocorre a mudança no valor da frequência de algumas instruções de um *trace* para outro.

Embora o tamanho do *trace* seja bastante eficiente em termos do tempo de simulação, o pequeno número e o reduzido tamanho das amostras provocou um desvio na frequência de instruções nos dois *traces*, ocasionando a maior diferença (cerca de 6%) nos dois *speed-ups* de todos os experimentos realizados (vide Tabelas 3 e 4).

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	0.23%	0.26%		0.32%	0.22%	
Mult PF	1.27%	0.30%		1.12%	0.62%	
Gráfica	0.73%			0.81%		
Memória	79.11%			87.08%		
Core	21.13%			9.71%	7.69%	5.86%

Tabela A.5: Ocupação das Unidades Funcionais (Espresso 1%)

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma	0.15%	0.16%		0.19%	0.16%	
Mult	0.53%	0.16%		0.56%	0.21%	
Gráfica	0.27%			0.30%		
Memória	79.30%			88.24%		
Core	22.60%			10.47%	8.21%	6.46%

Tabela A.6: Ocupação das Unidades Funcionais (Espresso 100%)

## A.5.2 Whetstone

	MÁQUINA A	MÁQUINA B
Tempo total:	58.866 ciclos	48.351 ciclos
<i>Speed-Up</i> :	2,52	3,06
Hit na BTB :	92,08 %	92,08 %

Tabela A.7: Estatísticas Whetstone 1%

A Figura 2 mostra a distribuição de instruções nos dois *traces* do programa Whetstone. Embora possamos observar pequenas diferenças entre as frequências das instruções, isso não compromete a qualidade da amostra.

Quando avaliamos a execução do *trace* amostrado nas arquiteturas A e B (Tabelas 7 e 8), podemos observar que existe uma diferença de cerca de 7,5% na taxa de acerto no BTB, em relação ao *trace* completo. Isto implica na pequena diferença entre o *speed-up* real e o amostrado (cerca de 2,6%), mas que afeta igualmente as duas arquiteturas, que apresentam valores comparativos muito próximos (21% e 22% de melhoria da arquitetura B sobre a A, para os *traces* amostrado e completo, respectivamente).

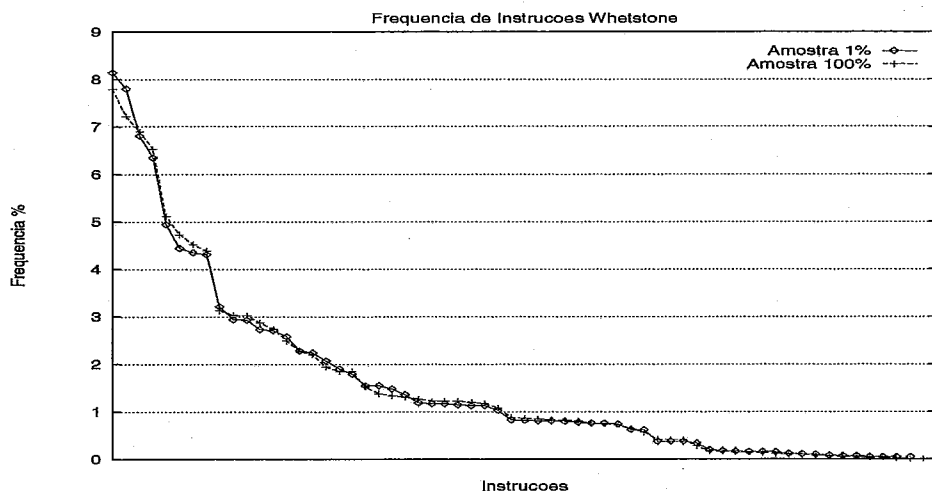


Figura A.2: Frequência das Instruções – Whetstone

	MÁQUINA A	MÁQUINA B
Tempo total:	5.741.995 ciclos	4.691.716 ciclos
<i>Speed-Up</i> :	2,57	3,14
Hit na BTB :	99,54 %	99,54%

Tabela A.8: Estatísticas Whetstone 100%

Em termos da ocupação das unidades funcionais (Tabelas 9 e 10), os valores encontrados apresentam diferenças absolutas de no máximo 1,75% (8% em termos relativos) o que é bastante razoável.

O *trace* amostrado do Whetstone é composto por 22 amostras, cada uma com 1.000 instruções. O número de amostras é aceitável, mas poderia ser melhor. A taxa de acerto no BTB foi também prejudicada pelo pequeno tamanho da amostra em relação ao tamanho da *cache*.

### A.5.3 Xlisp

A distribuição de instruções dos *traces* produzidos a partir do programa *xlisp* (vide Figura 3) apresenta uma diferença inferior a 3%. Além disso, não há alterações de ordem no histograma para as instruções mais executadas do programa (88%).

Examinando as Tabelas 11 e 12, podemos observar pequenas diferenças de desempenho provocadas pelos dois *traces* em cada arquitetura (configurações A e B). Contudo, quando comparamos as duas configurações, podemos verificar que elas apresentam a mesma taxa de falhas no BTB e *speed-up* relativo bastante próximo

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	12,39%	6,25%		13,71%	8,98%	
Mult PF	15,72%	11,36%		19,02%	13,96%	
Gráfica	15,57%			18,96%		
Memória	52,99%			64,52%		
Core	22,50%			12,83%	8,89%	5,67%

Tabela A.9: Ocupação das Unidades Funcionais (Whetstone 1%)

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	13,53%	7,04%		15,17%	10,01%	
Mult PF	17,16%	12,30%		20,76%	15,30%	
Gráfica	16,50%			20,20%		
Memória	52,00%			63,65%		
Core	22,33%			12,82%	8,91%	5,60%

Tabela A.10: Ocupação das Unidades Funcionais (Whetstone 100%)

	MÁQUINA A	MÁQUINA B
Tempo total:	462.233 ciclos	425.144 ciclos
<i>Speed-Up</i> :	2,11	2,29
Hit no BTB :	95,54%	95.54%

Tabela A.11: Estatísticas Xlisp 1%

(1,09 versus 1,095).

Quanto ao nível de ocupação das unidades funcionais, os valores encontrados são bem próximos, resultado talvez de uma distribuição equivalente das instruções, para os dois *traces* (vide Tabelas 13 e 14).

O *trace* do programa *xlisp* contém 160 amostras, cada uma com 1.000 instruções. Conforme ilustrado na Figura 3, o grande número de amostras permitiu caracterizar bem o programa, em relação a frequência de execução de cada instrução.

Como anteriormente, o pequeno tamanho das amostras influenciou a taxa de acerto do BTB e da *cache* (ocorreu uma pequena redução nas taxas de acerto). Talvez isto explique o *speed-up* ligeiramente menor do *trace* amostrado em relação ao *trace* completo.

Um dos objetivos propostos é o de estudar o impacto do uso do *trace* amostrado

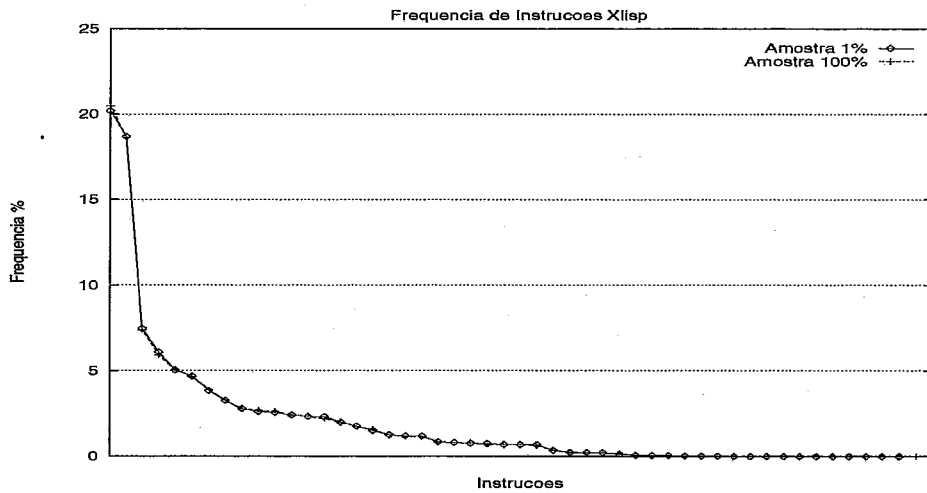


Figura A.3: Frequência das Instruções – Xlisp

	MÁQUINA A	MÁQUINA B
Tempo total:	44.087.642 ciclos	40.329.003 ciclos
Speed-Up:	2,20	2,41
Hit no BTB :	97,86%	97,86%

Tabela A.12: Estatísticas Xlisp 100%

na avaliação de mudanças arquiteturais. Neste sentido, a melhoria de desempenho que a arquitetura B apresenta sobre a arquitetura A é quase a mesma, utilizando o *trace* amostrado (9%) ou completo (9,05%).

#### A.5.4 Tomcatv

No programa tomcatv, a maior diferença absoluta entre as instruções do *trace* amostrado e do completo é inferior a 0,14%; e a maior diferença relativa não é superior a 2,5% para as instruções que somam 80% do total de instruções executadas no programa. O resultado pode ser visto na Figura 4.

Quando comparamos as arquiteturas A e B (vide Tabelas 15 e 16), podemos observar que os dois *traces* apresentam as mesmas diferenças de *speed-up* e de taxa de acerto no BTB. A arquitetura B apresentou um desempenho que é 12,7% superior ao da arquitetura A para os dois *traces*, e as taxas de acerto no BTB das duas configurações são idênticas quando do processamento do mesmo *trace*.

Conforme listado nas Tabelas 17 e 18, as taxas de ocupação das unidades funcionais de cada máquina são muito semelhantes para cada *trace*.



	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	0,02%	0,02%		0,02%	0,02%	
Mult PF	0,03%	0,02%		0,03%	0,03%	
Gráfica	0,01%			0,01%		
Memória	86,53%			94,07%		
Core	26,78%			11,98%	9,72%	7,42%

Tabela A.13: Ocupação das Unidades Funcionais (Xlisp 1%)

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	0,04%	0,05%		0,05%	0,04%	
Mult PF	0,07%	0,05%		0,07%	0,02%	
Gráfica	0,02%			0,02%		
Memória	86,17%			94,20%		
Core	27,94%			12,47%	10,29%	7,79%

Tabela A.14: Ocupação das Unidades Funcionais (Xlisp 100%)

	MÁQUINA A	MÁQUINA B
Tempo total:	1.605.432ciclos	1.423.473 ciclos
<i>Speed-Up</i> :	2,99	3,37
Hit no BTB :	99,82 %	99,82 %

Tabela A.15: Estatísticas Tomcatv 1%

O *trace* do programa Tomcatv contém 75 amostras, cada uma com 100.000 instruções. Estas são as condições ideais para a obtenção de um *trace* amostrado, já que o número de amostras é bastante acima daquele considerado ideal (em torno de 35), e o tamanho dos *traces* é consideravelmente maior que o tamanho da *cache* (400KB x 8KB), o que reduz as variações na taxa de acerto na *cache* e no BTB.

O tamanho deste *trace* (i.e., número e tamanho de amostras) associado a uma distribuição representativa de instruções, permite substituir o *trace* completo pelo amostrado, sem qualquer perda de qualidade nos resultados.

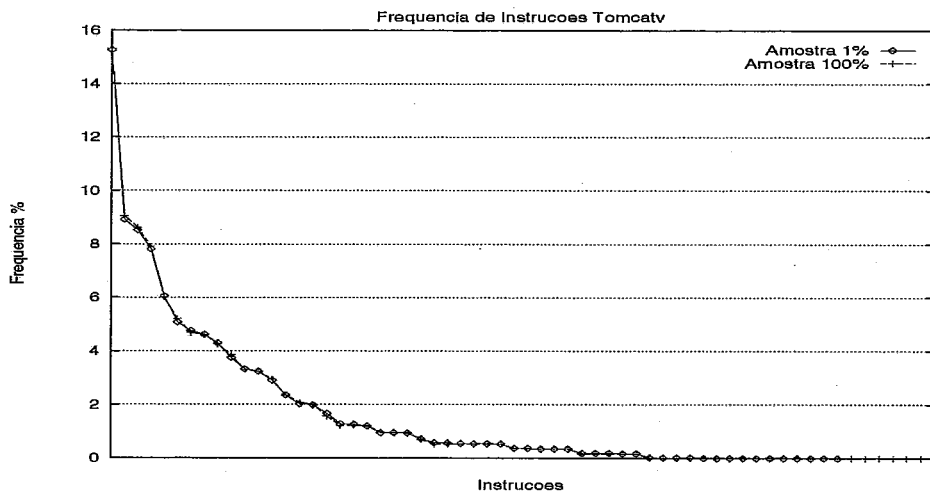


Figura A.4: Frequência das Instruções – Tomcatv

	MÁQUINA A	MÁQUINA B
Tempo total:	158.943.005 ciclos	141.014.855 ciclos
Speed-Up:	3,01	3,39
Hit no BTB :	100.00 %	100.00 %

Tabela A.16: Estatísticas Tomcatv 100%

## A.6 Conclusões

Intuitivamente a eficiência do *trace* amostrado é justificada pela grande diferença entre os tamanhos do código objeto e do *trace* completo de cada programa. Isto indica a presença de *loops* de instruções operando sobre vários conjuntos de dados.

A avaliação de uma arquitetura com *trace* amostrado será bem sucedida se este possuir características similares ao *trace* completo. Trabalhos anteriores sugerem o uso de diversos indicadores [MAR93] para caracterizar a qualidade do *trace* amostrado. Dentre esses, escolhemos em nosso trabalho o uso do IPC e da frequência de execução das instruções.

Um número maior de amostras permite obter frequências de instruções bem próximas nos dois tipos de *traces*. Caso necessário, o número de amostras pode ser aumentado até que as frequências obtidas sejam semelhantes. Na escolha do tamanho de cada amostra deve-se levar em conta o tamanho das *caches* simuladas. O tamanho das amostras pode ser dilatado, se o tamanho das *caches* simuladas for grande e se o IPC amostrado divergir em demasia. Caso o tamanho das amostras aumente proibitivamente, recomendamos como alternativa a reconstituição do estado

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	17,6%	9,5%		16,6%	13,9%	
Mult PF	19,2%	11,5%		20,0%	14,6%	
Gráfica	9,2%			10,3%		
Memória	74,7%			84,3%		
Core	27,0%			13,6%	9,9%	7,0%

Tabela A.17: Ocupação das Unidades Funcionais (Tomcatv 1%)

	MÁQUINA A			MÁQUINA B		
	F(1)	F(2)	F(3)	F(1)	F(2)	F(3)
Soma PF	17,6%	9,4%		16,5%	14,0%	
Mult PF	19,7%	11,9%		20,5%	15,1%	
Gráfica	9,1%			10,2%		
Memória	74,4%			83,9%		
Core	27,3%			13,7%	9,9%	7,1%

Tabela A.18: Ocupação das Unidades Funcionais (Tomcatv 100%)

da *cache* no início de cada amostra.

As dificuldades encontradas na geração e armazenamento dos *traces* completos, que consomem longos tempos de simulação e espaço em disco, limitaram o número de experimentos. Apesar disto, os resultados apresentados validam a proposta do uso de *trace* amostrado para comparar diferentes configurações super escalares.

Quando da avaliação do desempenho (*speed-up*) de uma classe de arquiteturas super escalares, a utilização de *traces* com 1% do tamanho original, provocou reduções nos tempos de simulação superiores a 100 vezes, com erros inferiores a 6,5%. Examinando a taxa de ocupação das unidades funcionais, verificamos que elas apresentaram erros aceitáveis (erros absolutos menores que 1,5%), garantindo desse modo uma avaliação segura da arquitetura em teste.

A redução nos tempos de simulação e de espaço em disco, eleva significativamente o número de alternativas que podem ser avaliadas pelo projetista da arquitetura super escalar.

## A.7 Bibliografia

- [CVE94] Cvetanovic, Z. and Bhandarker, D. "Characterization of Alpha AXP Performance Using TP and SPEC Workloads", Proceedings of the ISCA94, pp. 60-70, April 1994.
- [INT91] Intel, "i860 XR 64-bit Microprocessor", Intel Corp., June 1991, 78pp.
- [FER93] Souza, Alberto F. "Avaliando Parâmetros de uma Arquitetura VLIW", Tese de Mestrado, COPPE/Sistemas, Abril de 1993.
- [HAO93] Hao, Hsing T. "Efeito da Predição de Desvios e da Interrupção Precisa no Desempenho de Processadores Super Escalares", COPPE/UFRJ, Julho 1993.
- [KES91] Kessler, R.E.; Hill, M.D. ; Wood, D.A. "A Comparison of Trace Sampling Techniques for Multi-Megabytes Caches", Technical Report 1048, University of Wisconsin, Computer Sciences Department, September 1991.
- [LAH88] Laha, S; Patel, J.K., IYER, R.K. "Accurate Low-Cost Methods for Performance Evaluation of Cache Memory Systems", IEEE Transactions on Computers, Vol 37, No 11, pp. 1325-1336, Nov. 1988.
- [LAU93] Lauterbach, G. "Accelerating Architectural Simulation by Parallel Execution of Trace Samples", Sun Microsystems, SMLI TR-93-2, December 1993.
- [LEE84] Lee, J.K.F.; Smith, A.J. "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, January 1984.
- [LIU93] Liu, L and Peir, J-K. "Cache Sampling by Sets", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol.1, No. 2, pp 98-105, June 1993.
- [MAR93] Martonosi, M; Gupta, A; Anderson, T. "Effectiveness of Trace Sampling for Performance Debugging Tools", In Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, June 1993.
- [PLE94] Pleszkun, A. R. "Techniques for Compressing Program Address Traces", Proceedings of the 24th International Symposium on Microarchitecture, San Jose, CA, pp. 32-39, 1994
- [SHI94] Shipnes, J. and Phillip, M. "The PowerPC Performance Modeling", Communications of the ACM, Vol. 37, No 6, pp 47-63, June 1994.
- [TOM67] Tomasulo, R.M. "An Efficient Algorithm for Exploiting Multiple Arithmetics Units", IBM Journal, January 1967.

## **Apêndice B**

# **An Analytical Time and Area Model for On-chip Multiported Memories**

# An Analytical Time and Area Model for On-chip Multiported Memories

Gabriel P. Silva<sup>+</sup>; Eliseu M. C. Filho<sup>#</sup>; Victor M. Goulart<sup>®</sup>; Vladimir C. Alves<sup>®</sup>

NCE<sup>+</sup> - COPPE Sistemas<sup>#</sup> - COPPE Elétrica<sup>®</sup>

Universidade Federal do Rio de Janeiro

P.O.Box 68511 Cep 21945-970 - Rio de Janeiro - Brasil

e-mail:gabriel@cos.ufrj.br

### **Abstract**

This paper presents an analytical model for the silicon area, access time and cycle time of on-chip multiported memories. This model can be used to predict the area and timing characteristics of register files, tables, queues and other similar structures usually found in the implementation of superscalar processors.

## B.1 Introduction

The tradeoff between architecture and implementation is a major issue in the design of a superscalar microprocessor. Architectural decisions aimed at improving the degree of machine parallelism can affect the cycle time, chip area and power dissipation. This problem occurs, for example, in connection with architectural components that are implemented as on-chip multiported memories. The data register file is a well-known case. By increasing the number of read and write ports of the register file, the designer allows the processor to dispatch and complete more instructions on each cycle. Nevertheless, this may have an adverse impact on the implementation factors above mentioned.

In order to evaluate such tradeoff, the designer needs tools to estimate the impact of architectural parameters on the implementation. This work, presents an analytical model for the estimation of the access time, cycle time and silicon area of on-chip multiported memories. Besides the register file, other architectural components implemented as multiported memories are instruction queues, register renaming tables and reorder buffers[1].

Time models have been developed in [2] and [3], and an area model has been described in [3]. Those previous works basically focused on the design of cache memories. Moreover, compared with those models, the model presented here has the following enhancements:

- The model allows multiple read and write ports. On the contrary, the works mentioned above consider just a single read and a single write port;
- Both single-ended and differential sense amplifiers are considered in the model. This is important in the context of complex designs, like superscalar microprocessors. In the studies mentioned, only differential sense amplifiers were adopted;
- Two different types of memory cells are used, while previous works considered just one type;
- Here, an integrated time-area model is provided. The previous works present either a time model or an area model.



The remainder of this paper is organized as follows. The components and the structural parameters of on-chip multiported memories are identified in Section B.2. The basic electrical elements used in the time model are characterized in Section B.3. The time model is developed in Section B.4, while the area model is built in Section B.5. Section B.6 provides experimental results comparing time and area characteristics of different multiported memory configurations. The conclusions are given in Section B.7.

## B.2 On-Chip Multiported Memories

### B.2.1 The Components

The main components comprising on-chip multiported memories are: memory cells, decoders, wordline drivers, precharge drivers, write buffers, multiplexors, sense amplifiers and output drivers. Most on-chip multiported memories are based on static memory cells. Figure B.1 shows the two types of static memory cells considered in this work. The multiported cells used in the model are derived from these two basic structures.

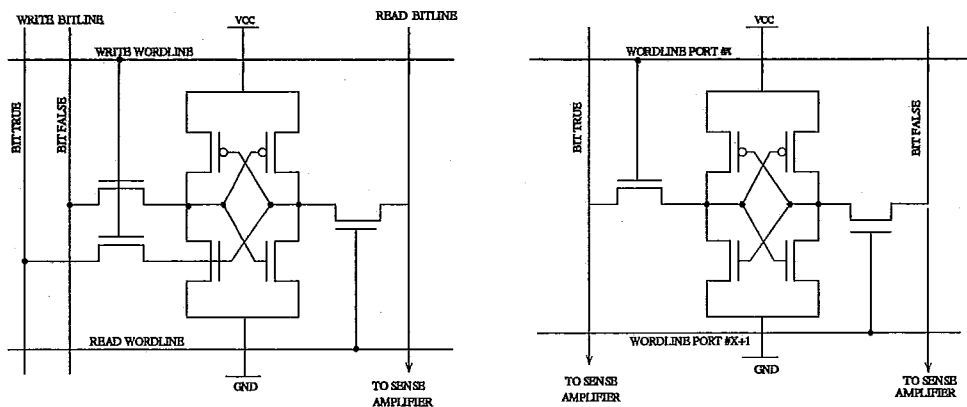


Figure B.1: Two Basic Memory Cells

The memory cell in Figure B.1(a) has distinct read and write bitlines and single-ended sense amplifiers for each bitline. In a read operation, the decoder selects a cell row by driving the corresponding read wordline high. For each memory cell in a row there is one read bitline which is initially precharged high. When the read wordline is activated, the value stored in the memory cell determines if the read bitline goes

high or low.

In the memory cell shown in Figure B.1(b), the bitlines are shared for both reading and writing. In this case, either one single-ended sense amplifier for each bitline or one differential sense amplifier for both bitlines can be employed. Differential sense amplifiers usually provide higher speed and better noise immunity [5], but it requires a reference voltage level and adds extra area. On the other hand, register files in superscalar processors usually exhibit a 2-to-1 ratio between the number of read ports and the number of write ports. A cell design with single-ended sense amplifiers can be a good choice, as it would require less area than a differential sense amplifier.

Regarding the write circuitry, a single-ended or differential scheme can be chosen for both cells shown in Figure B.1. Although there are some designs that adopt a non-differential circuit [6], the most reliable and commonly used circuit is the differential one.

In the layout of the cell, the designer can choose among several techniques to maximize speed and minimize noise. Some examples of techniques are: sharing cell diffusion contacts between adjacent cells in order to reduce bitline load; minimization of bitline metal length and maximization of bitline interspace, in order to reduce line-to-line coupling; and running ground lines along two adjacent rows of cells. The applicability of a particular technique depends on the number of metal layers available in the fabrication process. The above mentioned techniques were employed in the layout of the cells used to obtain the empirical results presented in Section B.6.

One additional optimization technique consists of splitting the memory cell array conveniently, allowing the same sense amplifier to be shared among several bitlines [2]. A multiplexor is inserted before the sense amplifier, with this multiplexor being controlled by the decoder. The number of read bitlines that share a sense amplifier depends on the value of the design parameters. This technique was also considered in the models.

## B.2.2 Memory Design Parameters

The following memory parameters are inputs to the time-area model: memory size in words, word size in bits, number of read ports and number of write ports. In addition, there are some internal parameters, which are discussed now.

In the cache organization presented by Wada [3], all memory cells in a line share a common wordline. Clearly, such an organization can lead to an inadequate aspect ratio, causing either the bitlines or wordlines to be very long and slow. This can result in a longer-than-necessary access time. In order to alleviate this problem, Wada describes how the array can be partitioned, and introduces two parameters. The  $N_{dwl}$  parameter indicates how many times the array is partitioned vertically, creating more, but shorter wordlines. The  $N_{dbl}$  parameter indicates how many times the array is partitioned horizontally, resulting in shorter bitlines. With this organization, the total number of subarrays is  $N_{dwl} \times N_{dbl}$ .

In [2], Wilton and Jouppi introduce another structural parameter, denoted by  $N_{spd}$ , which indicates how many cache lines are mapped to a single wordline. This allows the overall access time of the array to be changed without breaking it into smaller subarrays. In this case, two or more memory lines can share the same wordline, being selected later by a column multiplexor. This measure reduces the size of the first stage of the address decoder, thus improving the access time.

The optimal values of  $N_{dwl}$ ,  $N_{dbl}$ , and  $N_{spd}$  are automatically calculated by the model's software implementation, and depend on the memory size and number of ports. Notice that varying these parameters impacts the final memory area. Increasing  $N_{dwl}$  means that more wordline drivers are needed, while increasing  $N_{dbl}$  or  $N_{spd}$  requires more column multiplexors. Furthermore, increasing the number of ports requires more sense amplifiers. Because we are modeling a memory (which is equivalent to a direct mapped cache), there is no need for a multiplexor to select the appropriate sense amplifier output.

Let  $M_{bytes}$  and  $W_{bytes}$  be the memory size in bytes and the wordline size in bytes, respectively. Using the above structural parameters, each subarray contains  $8 \times W_{bytes} \times (N_{spd}/N_{dwl})$  columns and  $(M_{bytes}/W_{bytes}) \times N_{dbl} \times N_{spd}$  rows. These expressions will be useful in the models developed in Sections B.4 and B.5.

## B.3 Modeling the Basic Parameters

As it will be discussed in Section B.4, the time model is obtained by decomposing each memory component into first-order equivalent RC circuits and taking into account the array structure and the characteristics of the fabrication process. Although its relatively simplicity, the model considers signal transition delays, what is essential for an accurate time estimations in submicron technologies. This section describes how the transistors and parasitics resistances and capacitances are modeled.

The RC approximations require the estimation of the full-on and the switching resistance of a transistor, denoted by  $R_{on}$  and  $R_{sw}$ , respectively. The parameter  $R_{on}$  is the drain-source resistance seen when the transistor is fully conducting and with a constant gate voltage. This definition is also used for pass transistors that are always conducting. It is assumed that  $R_{on}$  and  $R_{sw}$  are inversely proportional to the transistor width,  $W$ .

The RC approximations also require the estimation of the transistor's gate capacitance,  $C_{GATE}$ , and of its drain capacitance,  $C_{DRAIN}$ . The gate capacitance of the transistor consists of the capacitance of the gate itself and the capacitance of the polysilicon line connecting the gate. The value of the gate capacitance itself depends on whether the transistor is a pass transistor, a pull-up or a pull-down transistor [2]. The gate capacitance  $C_{GATE}$  is given by:

$$(1) \quad C_{GATE} = (W \times L_{eff} \times C_{gate}) + (L_{poly} \times L_{eff} \times C_{poly})$$

where  $W$  is the transistor width;  $L_{eff}$  is the effective length of the transistor;  $C_{gate}$  is the capacitance of the gate itself per unit area;  $L_{poly}$  is the length of the polysilicon line connecting the gate; and  $C_{poly}$  is the polysilicon line capacitance per unit area. This formula holds for both NMOS and PMOS transistors.

The drain capacitance depends on the drain's area and perimeter. Two equations are used to describe the drain capacitance: one is applicable if the transistor width  $W$  is less than  $10\mu m$ , and other if the width is greater than this value. For the first case, the drain capacitance expression is:

$$(2) \quad C_{DRAIN} = (3 \times W \times L_{eff} \times C_{diffarea}) + ((6 \times L_{eff} + W) \times C_{diffside}) + (W \times C_{diffgate})$$

where  $C_{diffarea}$ ,  $C_{diffside}$  and  $C_{diffgate}$  are process dependent parameters [2] having different values for NMOS and PMOS transistors. If the width is greater than  $10\mu m$ , it is assumed that the transistor is folded, reducing the drain capacitance. In this case, the drain capacitance is given by the following expression:

$$(3) \quad C_{DRAIN} = (3 \times L_{eff} \times W/2 \times C_{diffarea}) + (6 \times L_{eff} \times C_{diffside}) + (W \times C_{diffgate})$$

The time model takes into account the existence of multiple ports. Each new port added to the memory will require additional wordlines, bitlines and pass transistors, thus affecting the resistance and capacitance associated with wordlines and bitlines. Also, there is an increase in the dimensions of the basic memory cell for each new wordline or bitline added. This increase depends on many factors, such as the number of metal available, the relative size of the metal lines compared to the transistors and the designer's skill. In [4], Mulder estimates a linear increase of about 25% in each dimension for each new metal line added, for technologies between  $1.0\mu m$  and  $2.0\mu m$  and 2-metal layers. Our experiments indicated a linear increase of 33% when using  $0.5\mu m$  and  $0.7\mu m$ , 2-metal layers technologies.

The time model also takes into account parasitic resistances and capacitances of the wordlines and bitlines. These resistances and capacitances have fixed values per unit length. In [2], Wilton and Jouppi used the memory cell shown in Figure 1(b), with differential sense amplifiers. To adapt the expressions developed in that work to multiported memories, we obtain:

Wordline resistance of a metal line per bit width:

$$(4) \quad R_{WORDMETAL} = R_{wordmetal}(1 + PORTFACTOR \times N_{bitl})$$

Wordline capacitance of a metal wire per bit width:

$$(5) \quad C_{WORDMETAL} = C_{wordmetal}(1 + PORTFACTOR \times N_{bitl})$$

Bitline resistance of a metal line per bit height:

$$(6) \quad R_{BITMETAL} = R_{bitmetal}(1 + PORTFACTOR \times N_{wordl})$$

Bitline capacitance of a metal wire per bit height:

$$(7) \quad C_{BITMETAL} = C_{bitmetal}(1 + PORTFACTOR \times N_{wordl})$$

Width of a memory cell:

$$(8) \quad Bit_{WIDTH} = Bit_{width}(1 + PORTFACTOR \times N_{wordl})$$

In expressions (4) to (8),  $N_{bitl}$  and  $N_{wordl}$  are, respectively, the number of extra bitlines and wordlines per memory cell, resulting from the addition of multiple ports (see expressions (17) and (18)). The parameter  $PORTFACTOR$  is an empirical scaling factor representing the effect of the inclusion of a new metal line (see Section B.5). The parameter  $Bit_{width}$  is the width of the single-ported, basic memory cell.

## B.4 The Time Model

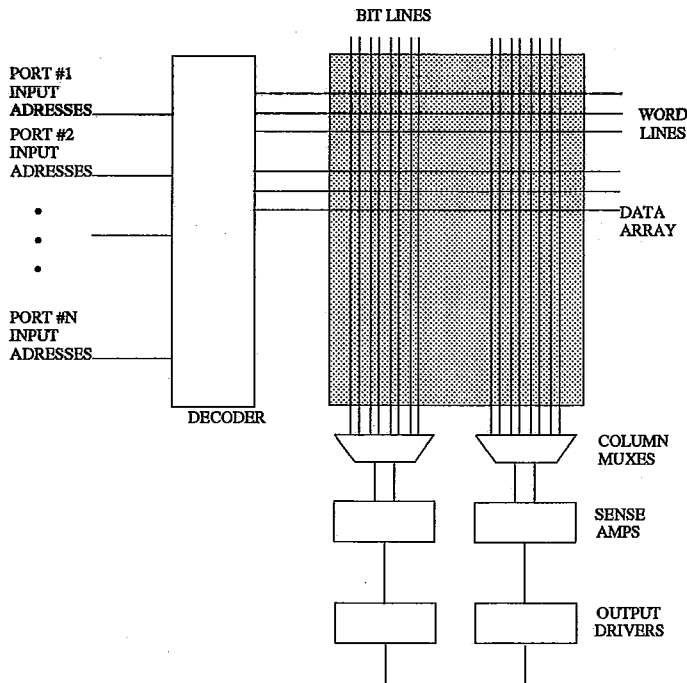


Figure B.2: Components of a Multiported On-chip Memory

The time model takes into account the delays introduced by the decoder, wordlines, bitlines, sense amplifiers and output drivers as shown in (Figure B.2). Both

memory cells shown in Figure B.1 are considered, as well as single-ended and differential sense amplifiers. The delay of each of these components is estimated separately and the results combined to obtain the access and cycle time of the multiported memory. These estimations will depend on the number of ports of the memory and the type of the sense amplifier employed. The following subsections show how the component models originally developed by Wilton and Jouppi in [2] are affected by the introduction of multiported memory cells.

### B.4.1 The Decoder

The model assumes a decoder composed by three stages. Each block in the first stage takes three address bits (in true and complement form) and generates a 1-out-of-8 code, driving a precharged decoder bus. The generated codes are combined using NOR gates in the second stage. The final stage is an inverter which drives each wordline driver.

Estimating the wire lengths in the decoder requires knowledge of the memory tile layout. As mentioned in Section B.2, the memory is divided into  $N_{dwl} \times N_{dbl}$  subarrays. Each of these arrays is  $8 \times W_{bytes} \times (N_{spd}/N_{dwl})$  cells wide. It is assumed that they are grouped in two-by-two blocks, with the 3-to-8 predecode NAND gates at the center of each block. This reduces the length of the connection between the decoder driver and the predecode block to approximately one quarter of the total memory width, or  $2 \times W_{bytes} \times N_{dbl} \times N_{spd}$ . The length of the connection between the predecode block and the NOR gate is then (on average) half of the subarray height, which is  $M_{bytes} \times W_{bytes} \times N_{dbl} \times N_{spd}$  cells. In large memories with many groups the bits in the memory are arranged so that all bits driving the same data output bus are in the same group, thus shortening the data bus.

Changing the number of memory ports will change the number of decoders, since an extra decoder is needed for each new port added (if it is not shared). This will have an impact in the total area, but a negligible change in the decoder's delay.

### B.4.2 Wordline Driver

In [3], the size of the wordline driver does not depend on the number of cells attached to the wordline. This severely overestimates the wordline delay of large arrays. Here,

it is assumed a variable-sized wordline driver, as was done by Wilton and Jouppi in [2]. Usually, the driver size is adjusted appropriately, according to a desired rise time.

For the wordline delay, the new values  $R_{WORDMETAL}$  and  $C_{WORDMETAL}$  (see expressions (4) and (5), respectively) must be used in order to take into account the effect of multiple ports. The modified expressions for the wordline resistance  $R_{eq}$  and capacitance  $C_{eq}$ , used to calculate the final wordline delay, are:

$$(9) \quad R_{eq} = R_{WORDMETAL} \times cols/2$$

$$(10) \quad C_{eq} = gatecap_{pass}(W_a, (BitWIDTH - 2 \times W_a)/2) + (C_{WORDMETAL} \times cols) + draincap(N_{size}) + draincap(P_{size})$$

where  $gatecap_{pass}()$  is a two-parameter function which gives the gate capacitance of the pass transistor of the memory cell;  $W_a$  is the width of the access transistor;  $draincap()$  is also a function which gives the drain capacitance for both transistors of the wordline driver;  $N_{size}$  and  $P_{size}$  are, respectively, the sizes of the NMOS and PMOS transistors; and  $cols$  is the number of columns of the cell array. In particular,  $(C_{WORDMETAL}) \times cols$  is the total capacitance of the metal wordline.

### B.4.3 Bitlines and Memory Cells

Wada's model does not apply to memories with column multiplexing. Wilton and Jouppi model allows column multiplexing using NMOS pass transistors between several pairs of bitlines and a shared sense amplifier. In their model, the degree of column multiplexing (number of pairs of bitlines per sense amplifier) is  $N_{spd} \times N_{dbl}$ . The bitlines are precharged with two NMOS diodes to less than  $V_{dd}$  since the differential sense amplifier performs poorly with a common-mode voltage of  $V_{dd}$ . We are considering the same voltage level also for the single-ended sense amplifier case.

None of those models considers the use of multiported memory cells. Depending on the memory cell design, the addition of one port will, at least, imply in the need of one extra wordline and one or two extra bitlines. According to the type of memory cell and sense amplifier employed, there are three cases to consider.



When using the memory cell shown in Figure B.1(a), each new write port that is added to the memory requires one wordline and two bitlines. Similarly, one wordline and one bitline are added for each new read port. This can be expressed as:

$$(11) \quad N_{wordlines} = N_{read} + N_{write}$$

$$(12) \quad N_{bitlines} = N_{read} + (2 \times N_{write})$$

where  $N_{read}$  and  $N_{write}$  are the number of read and write ports, respectively; the parameters  $N_{wordlines}$  and  $N_{bitlines}$  are, respectively, the number of wordlines and bitlines.

When using the memory cell shown in Figure B.1(b), for each new write port added, the number of wordlines and bitlines increases by one and two, respectively. With the addition of a new read port, the situation becomes more complex, since wordlines and bitlines can be shared, and single-ended or differential sense amplifiers can be used.

If differential sense amplifiers are employed, for each new read port added, one wordline and two bitlines must be added, *only* if the number of read ports is greater than the number of write ports. Otherwise, the wordlines and bitlines of the new port can be shared with those of an existing write port. See the following expressions:

$$(13) \quad N_{wordlines} = MAX(N_{read}, N_{write})$$

$$(14) \quad N_{bitlines} = MAX(2 \times N_{read}, 2 \times N_{write})$$

If single-ended sense amplifiers are used, two situations arise when a new port is added to the memory: (1) if the number of read ports is greater than twice the number of write ports, the number of wordlines and bitlines will increase by one unit; (2) in the other case, the wordlines and bitlines of the new read port can be shared with existing wordlines and bitlines of a write port. The expressions below calculate the number of bitlines and wordlines in this case:

$$(15) \quad N_{wordlines} = MAX(N_{read}, ceil(N_{read} + (2 \times N_{write})/2))$$

$$(16) \quad N_{bitlines} = MAX(N_{read}, 2 \times N_{write});$$

Finally, the number of additional wordlines  $N_{wordl}$  and the number of additional bitlines  $N_{bitl}$  are given by:

$$(17) \quad N_{wordl} = N_{wordlines} - 1;$$

$$(18) \quad N_{bitl} = N_{bitlines} - 2;$$

The addition of either a new wordline or a bitline has an impact on the memory layout and in the final memory performance. As each new (metal) line is added, the memory array dimensions increase, as well as its corresponding capacitance and resistance. This will affect directly wordline and bitline delays. Also, some transistor dimensions must be altered in order to support the new pass transistors added for each read/write port. Particularly, the pull-up and pull-down transistors of the memory cell are increased in their sizes, in order to avoid a destructive read of the memory cell. The critical sizing ratio in the cell is between the parallel combination of pass transistors and the N-channel pull-down of the static memory cell. A minimum ratio of 2 is a safe target [5].

### Adapting the Expressions

The equivalent circuit used to calculate the bitline and memory cell delays comprises four components:  $R_{mem}$  in parallel with  $C_{line}$  in parallel with the combination of  $C_{column}$  in series with  $R_{line}$ . Parameter  $R_{mem}$  is the equivalent resistance of the conducting transistors in the memory cell and remains constant with the use of multiple ports. Parameter  $C_{line}$  includes: the capacitance of the memory cells sharing the bitline, the metal line capacitance, the drain capacitance of the precharge circuit ( $draincap(W_{equ})$ ), and the drain capacitance of the column multiplexor of the pass transistor ( $draincap(W_{bitmux})$ ). The expression for  $C_{line}$  is:

$$(19) \quad C_{line} = rows \times (C_{bitrow} + C_{BITMETAL}) + 2 \times draincap(W_{equ}) + draincap(W_{bitmux})$$

where  $C_{bitrow}$  is the bitline capacitance of the memory cell. This parameter remains constant with the introduction of multiple ports, since the transistors between  $V_{dd}$  and the bitline do not change in size. On the contrary,  $C_{BITMETAL}$  (see expression (7)) changes its value as the memory cell's dimension increase with the addition of multiple ports.

$C_{colmux}$  is the capacitance seen by the output of the conducting column pass transistor. It includes the drain capacitance of all pass transistors connected to this sense amplifier and the input capacitance of the sense amplifier. Thus,  $C_{colmux}$  depends on the type of sense amplifier, the expression is:

$$(20) \quad C_{colmux} = 2 \times gatecap(W_{senseinv})$$

where  $gatecap(W_{senseinv})$  is the gate capacitance of the differential sense amplifier's input transistor.

With a single-ended amplifier, capacitance  $C_{colmux}$  increases, since there is an extra pass transistor (as shown in the next section). The expression for this case is:

$$(21) \quad C_{colmux} = 2 \times gatecap(W_{senseinv}) + draincap(W_{sensepass})$$

where  $gatecap(W_{senseinv})$  is the gate capacitance of the single-ended sense amplifier's input transistor, and  $draincap(W_{sensepass})$  is the drain capacitance of the extra pass transistor.

$R_{line}$  is the resistance of the bitline resistance. This resistance increases with the number of ports because the memory cell has a correspondent increase in its dimension. The expression for  $R_{line}$  is:

$$(22) \quad R_{line} = R_{BITMETAL} \times rows/2$$

### B.4.4 Sense Amplifiers and Output Drivers

There are two basic sensing schemes of the memory cell: differential (see and single-ended. A differential sensing scheme, as shown in Figure B.4, usually provides better speed and noise immunity [5], but there is the need to generate a reference level, which could be difficult under certain circumstances and costs extra silicon area. The single-ended scheme, seen in Figure B.3, needs around 80% more time to sense the memory cell [5], but is simpler and smaller. Some circuits [6] could be employed to enhance its noise immunity, at the expense of some area. The delay is approximated by a constant, which is obtained through spice simulations for both types of sense amplifiers.

The output driver for a multiported memory is simpler than the one used for a cache [2]. The output driver delay also changes because the memory cell changes its dimension with the number of ports.

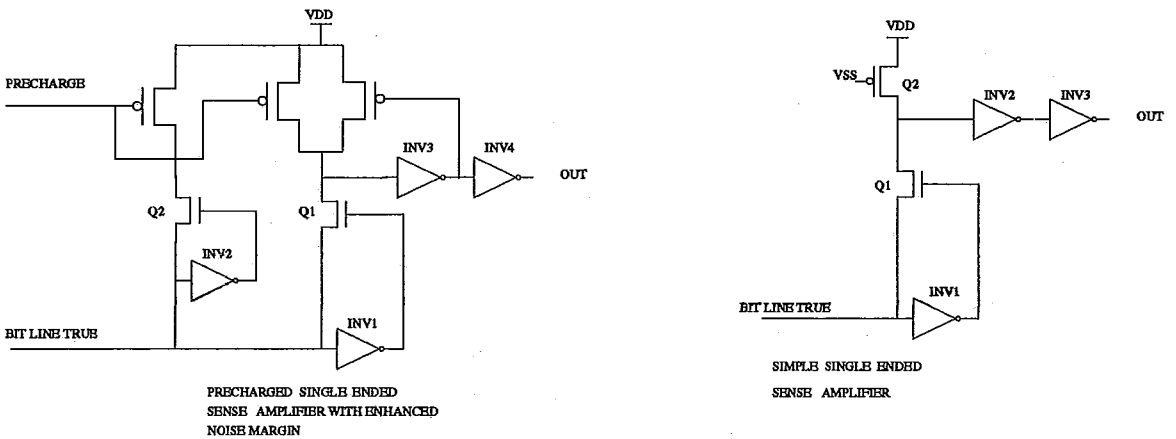


Figure B.3: Single-ended Sense Amplifiers

The output driver delay also changes because the basic memory cell changes its dimension with the number of ports, modifying the values of the capacitances and resistances seen by it. Thus, the expression:

$$(23) \quad C_{eq} = (\text{draincap}(W_{outN}) + \text{draincap}(W_{outP})) + C_{WORDMETAL} \times (8 \times W_{bytes} \times N_{spd} \times (vstack)) + C_{out}$$

$$(24) \quad R_{eq} = R_{WORDMETAL} \times ((8 \times W_{bytes} \times N_{spd} \times (vstack))/2)$$

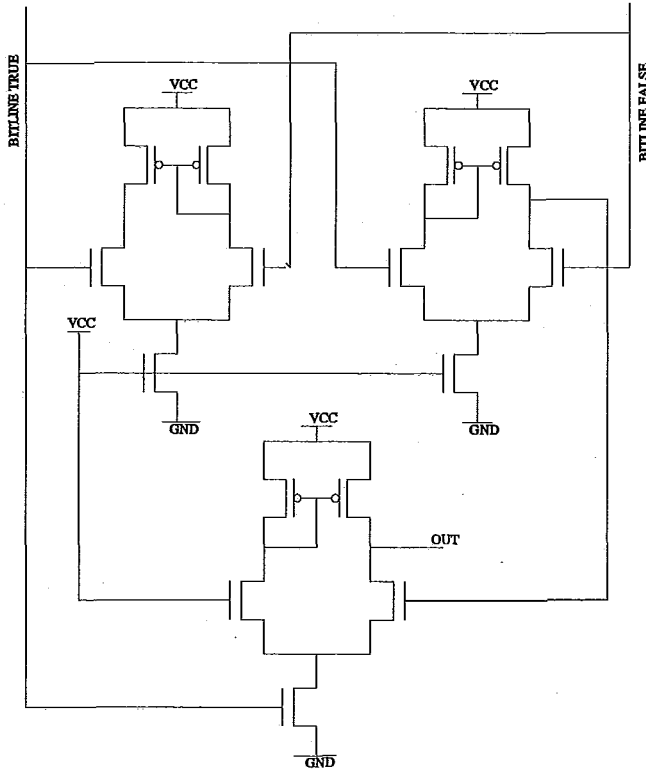


Figure B.4: Differential Sense Amplifier

where:  $draincap(W_{out})$  is a function which calculates the drain capacitance of the output driver's transistors (N and P transistors);  $C_{out}$  is the output capacitance of the memory and  $vstack$  is the number of arrays stacked vertically, resulting from the memory array partitioning.

### B.4.5 Total Access and Cycle Time

The access time can be derived from the following expression:

$$(25) \quad T_{acc} = T_{decoder} + T_{wordline} + T_{bitline} + T_{sense} + T_{driver}$$

The model first attempts to find the array organization parameters that resulted in the lowest access time via exhaustive search for each memory parameter,  $N_{spd}$ ,  $N_{dwl}$  and  $N_{dbl}$ . The difference between the access and cycle time of a memory varies widely depending on the circuit techniques used. We have chosen to model a conventional structure with the cycle time equal to the access time plus the precharge.

There are three elements in our assumed memory organization that need to be

precharged: the decoders, the bitlines, and the comparator. The precharge times for these elements are somewhat arbitrary, since the precharging transistors can be scaled in proportion to the loads they are driving. It is assumed that the time for the wordline to fall and bitline to rise in the data array is the dominant part of the precharge delay. Given properly dimensioned transistors in the wordline drivers, the wordline fall time is approximately the same as the wordline rise time. It is assumed that the bitline precharging transistors are scaled such that a constant (over all memory organizations) bitline charge time is obtained. This constant is, of course, technology dependent. Supposing that this constant is equal to four inverter delays (each with a fanout of four, [2]). Thus, the cycle time of the cache is given by:

$$(26) \quad T_{memory} = T_{acc} + T_{wordlinedelay} + 4 \times (inverterdelay)$$

## B.5 The Area Model

For the area model, it was considered a memory cell with one wordline and two bitlines, with dimensions  $Bit_{Height}$  and  $Bit_{Width}$ , respectively. The number of bitlines and wordlines increase as well as the number of ports increase. The empirical factor, PORTFACTOR, reflects the increase of each dimension of the memory cell for each additional wordline and bitline. The expression for the memory cell area is:

$$(27) \quad A_{cell} = (1+PORTFACTOR) \times N_{bitl} \times Bit_{width} \times (1+PORTFACTOR) \times N_{wordl} \times Bit_{Height}$$

where  $N_{wordl}$  and  $N_{bitl}$  are given by Equations (17) and (18), respectively. As mentioned, a PORTFACTOR of 33% was obtained for cell layouts designed using 0.5  $\mu m$  and 0.7  $\mu m$ , 2-metal layers technology. Given the individual memory cell area, the total array area is simply:

$$(28) \quad A_{array} = A_{cell} \times N_{words} \times 8 \times W_{bytes}$$

where  $N_{words}$  is the number of words in the array and  $8 \times W_{bytes}$  is the output width, in bits.

There are many aspects to be considered in the area estimation. First, the

number of decoders used is proportional to the number of data subarrays, and the size of each decoder is proportional to the size of the whole array. This cell array organization has also a direct impact on the wordline driver, which is wider as longer are the metal wordlines. The number of sense amplifiers depends on the sensing scheme and on the number of bitlines. Recall that the area occupied by a single-ended sense amplifier is half the area occupied by a differential sense amplifier, but the latter is faster than the first one. Depending on the memory organization parameters, the area due to the column multiplexor need also to be added, just a pass gate and respective wiring for each column. All those aspects were take into account in our area model as required for an accurate estimation.

The write buffer ( $Wrbuf$ ) and precharge ( $Prechg$ ) circuitry areas are dominated by the transistor size, rather than wiring and component spacing. This area is approximated by the following expressions:

$$(29) \quad Wrbuf_{size} = 4 \times W_{equ} \times L_{eff} \times ROUTEFACTOR$$

$$(30) \quad Prechg_{size} = 7 \times W_{equ} \times L_{eff} \times ROUTEFACTOR$$

$$(31) \quad Prechg_{area} = Prechg_{size} \times (N_{bitlines}/2) \times cols + (Wrbuf_{size} \times N_{write} \times cols)$$

where  $W_{equ}$  is the width of the equilibration transistor of the precharge circuit;  $L_{eff}$  is the effective length of the transistor and ROUTEFACTOR is a empirical factor which reflects the additional area required for routing. In this work, it was adopted a ROUTEFACTOR of 1.5, based in our experiments.

The wordline driver is as larger as wider is the memory subarray it is driving. Thus, its size  $W_{drvsiz}$  varies with the number of subarrays and should be recalculated at each iteration, for each new value of  $N_{dbl}$ ,  $N_{dwl}$  and  $N_{spd}$ :

$$(32) \quad W_{drvsiz} = (((P_{size} + N_{size}) \times L_{eff}) + ((W_{decinvP} + W_{decinvN}) \times L_{eff})) \times ROUTEFACTOR$$

where  $W_{decinv}$  is the width of the decoder's inverter transistors. The total wordline

driver area ,  $A_{wordrv}$ , can be expressed by:

$$(33) \quad A_{wordrv} = W_{drvsiz} \times N_{wordlines} \times rows$$

The decoder area varies with the number of subarrays and the number of rows of each subarray. There are as many decoder drivers (area in equation (35)) as the number of address bits plus two, since we need true and inverted address lines:

$$(34) \quad Decdrv_{size} = ((W_{decdriveP} + W_{decdriveN}) \times L_{eff}) \times ROUTEFACTOR$$

$$(35) \quad A_{decdrv} = Decdrv_{size} \times 2 \times \log_2(N_{bytes}/W_{bytes})$$

Each NAND in the second stage of the decoder has three inputs and there are 8 NANDs per block, and there are a total of  $numstack$  blocks in each subarray [2]. There are  $N_{dwl} \times N_{dbl}$  subarrays, so we can estimate the area of this decoder stage as:

$$(36) \quad W_{dec3to8}_{size} = ((W_{dec3to8N} + W_{dec3to8P}) \times L_{eff}) \times ROUTEFACTOR$$

$$(37) \quad A_{dec3to8} = 8 \times 3 \times Dec3to8_{size} \times numstack \times N_{dwl} \times N_{dbl}$$

In the last stage of the decoder, there is one NOR driver per wordline in every subarray, which area is expressed by (37). Each of these NOR has  $numstack$  inputs, given a total area in (42):

$$(38) \quad Decnor_{size} = ((W_{decnorN} + W_{decnorP}) \times L_{eff}) \times ROUTEFACTOR$$

$$(39) \quad A_{decnor} = numstack \times Decnor_{size} \times (M_{bytes}/W_{bytes}) \times N_{wordlines}$$

It is also necessary to consider the output driver area, one driver for each output bit of the memory array:

$$(40) \quad Outnor_{size} = (W_{outnorN} + W_{outnorP}) \times L_{eff} \times ROUTEFACTOR$$



$$(41) \quad Outdrv_{size} = (W_{outdrvN} + W_{outdrvP}) \times L_{eff} \times ROUTEFACTOR$$

$$(42) \quad A_{out} = (2 \times Outnor_{size} + Outdrv_{size}) \times 8 \times W_{bytes}$$

The total area of the memory device is the sum of all these stages, as shown below:

$$(43) \quad A_{total} = A_{decnor} + A_{dec3to8} + A_{decdrv} + A_{wordrv} + A_{prechg} + A_{array} + A_{sense} + A_{out}$$

The most important characteristic of the area model is that it considers the impact of multiple ports in the memory cell layout, and it allows quick and accurate comparisons between different implementation alternatives.

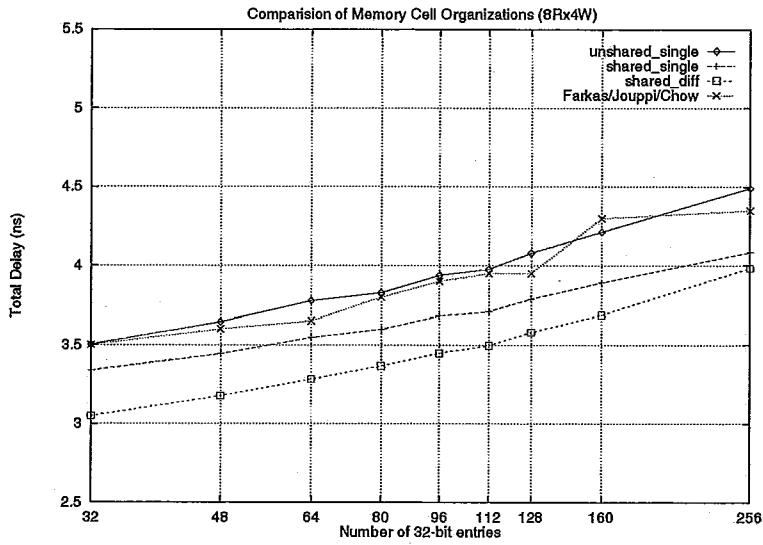
## B.6 Experimental Results

The time-area model just described assumes that the dimensions of the memory cell transistors do not change as the design scales up. Faster memories could be obtained by increasing memory cell transistor sizes and/or wordline drivers, but this would have an area penalty in the final design, possibly resulting in a prohibitive size as the number of words and/or ports increase.

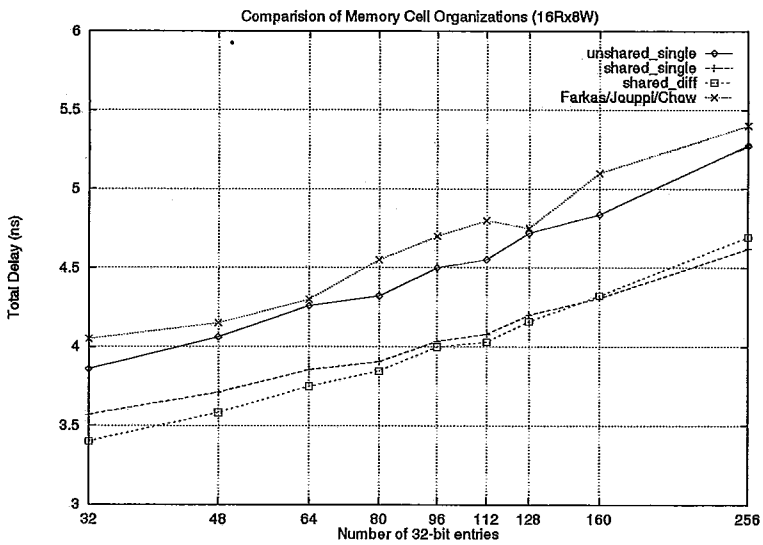
This section shows time and area estimations for multiported memories to be used as register files in superscalar processors. In all configurations considered, there are 32-bit memory words, and the number of read ports is twice the number of write ports, as it is usually the case in superscalar processors.

### B.6.1 Cycle Time Analysis

The plots in Figure B.5 show the total cycle time of the memory versus the number of words for two port configurations: 8 read ports and 4 write ports (Figure B.5(a)); and 16 read ports and 8 write ports (Figure B.5(b)). Both plots were obtained by simulating a memory with three different types of memory cells: (1) the memory cell shown in Figure B.1(a), with distinct wordlines and bitlines and single-ended sense amplifiers; (2) the memory cell shown in Figure B.1(b), with shared wordlines



(a)



(b)

Figure B.5: Time Delay for Two Memory Configurations

and bitlines and differential sense amplifiers; (3) the memory cell shown in Figure B.1(b), but with shared wordlines and bitlines and single-ended sense amplifiers.

In [8], Farkas *et al.* considered identical port configurations in their model analysis, with a memory cell identical to the one shown in Figure B.1(a). Our plotting shows that our model is very similar to theirs, with minimal delay difference, using the same memory cell organization (see `unshared_single` curve).

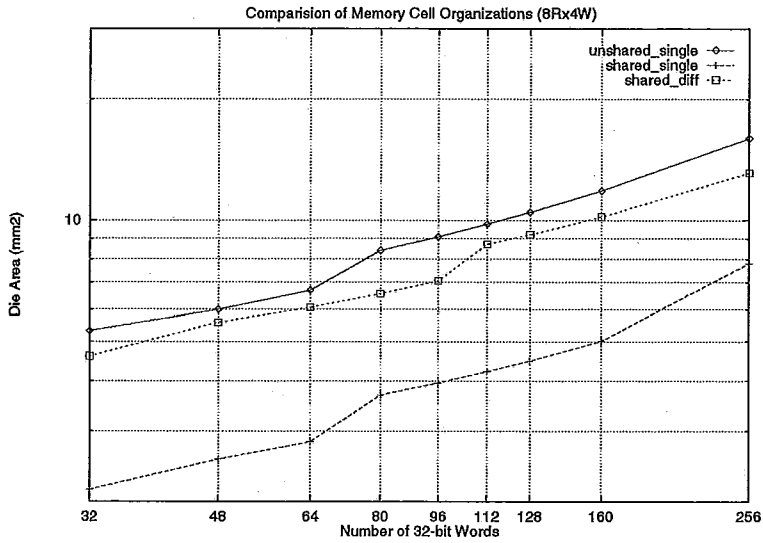
Using the memory cell of Figure 1(b) the results are significantly improved. The memory cell using differential sense amplifier (`shared_diff` curve) has a better performance due to its faster sensing scheme. But, although the single-ended amplifier is slower, the number of bitlines and wordlines per memory cell required is smaller, and the result with the `shared_single` memory cell is practically identical to the obtained with the `shared_diff` cell.

## B.6.2 Area Analysis

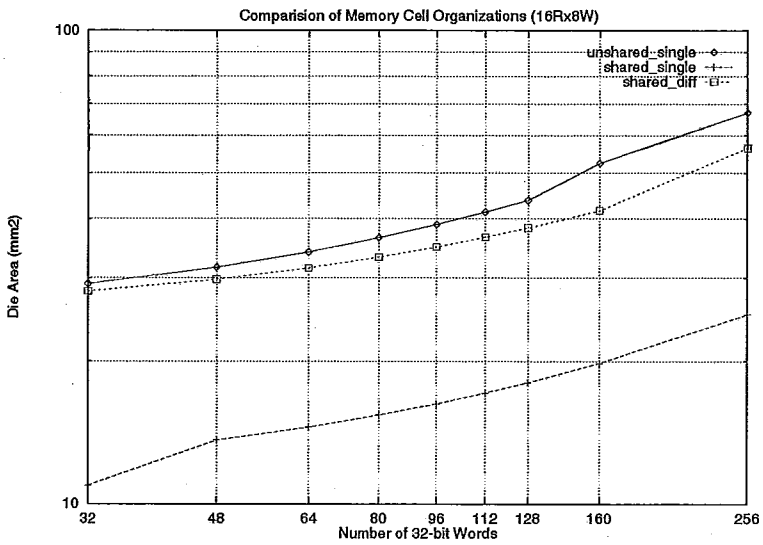
Area estimations for the memory organizations analyzed in the previous section are summarized in Figure B.6 using all possible combinations of basic memory cells and sense amplifiers for two memory port configurations: 8 read ports and 4 write ports (Figure B.6(a)); 16 read ports and 8 write ports (Figure B.6(b)). The plots show the total memory area versus the number of words (32 bits). They indicated that doubling the number of words is less significant in terms of area than doubling the number of ports. We found that the `unshared_single` and the `shared_diff` memory cells are the most area consumer. The `shared_single` memory cell presents the best time/area relation among all three memory types.

Mulder [4] presented an area model and checked it against some designs using old technologies. This does not apply to the model presented here, which models submicron technology. We checked our model with the design in [5], and our estimations perform very well. We predicted a total area of 1.2 mm<sup>2</sup>, and they claimed a total area of 1.4 mm<sup>2</sup>, that included some dedicated registers.

We present an open model, so the designer can modify it according to his/her needs. Our experiments show that the model is accurate concerning comparisons made within the same technology parameters.



(a)



(b)

Figure B.6: Area Estimations for Two Memory Configurations

## B.7 Conclusions

We have presented an analytical model for both the area, access and cycle times of embedded multiported memories. The advantage of using an analytical model is that its computational complexity is considerably lower than that of an electrical simulation. These results and the constraints associated with each alternative can be rapidly evaluated by the designer. Despite previous analytical models, ours has integrated three basic multiported memory cell designs and two types of sense amplifiers in both time and area models. In order to achieve higher accuracy these models consider: non-step stage input slopes; rectangular stacking of memory sub-arrays; a transistor-level decoder model; column-multiplexed bitlines; modern array organization parameters and load-dependent transistor sizes for wordline drivers.

## B.8 Bibliography

- [1] G. S. Sohi, J. E. Smith, "The microarchitecture of superscalar processors", Proceedings of the IEEE, V. 83, N. 12, pp. 1609-1624, Dec. 1995
- [2] S. J. Wilton, N. P. Jouppi, "An access and cycle time model for on-chip cache", Technical Report 93/5, DEC - Western Research Lab, July 1994.
- [3] T. Wada, S. Rajan, S. A. Przybylski, "An analytical access time model for on-chip cache memories", *IEEE Journal of Solid-State Circuits*, V. 27, N. 8, pp 1147-1156, Aug. 1992
- [4] J. M. Mulder, N. T. Quach, M. J. Flynn, "An area model for on-chip memories and its application", *IEEE Journal of Solid-State Circuits*, V. 26, N. 2, pp 98-106, Feb. 1991.
- [5] R. D. Jolly, "A 9-ns, 1.4 Gbytes/s, 17-Ported CMOS Register File", *IEEE Journal of Solid State Circuits*, V. 26, N. 10, pp 1407-1412, Nov. 1991.
- [6] L. A. Lev, et al. "A 64-bit Microprocessor with Multimedia Support", *Journal of Solid State Circuits*, V. 30, N. 11, pp 1127-1238, Nov. 1995
- [7] S. J. Wilton, N. P. Jouppi, "Timing Models for MOS circuits" Technical Report SEL 83-03. Interated Circuits Laboratory, Stanford University, 1983
- [8] K. I. Farkas; N. P. Jouppi, P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors", Technical Report 95/10, DEC - Western Research Lab, Nov. 1995.

# Apêndice C

## Proposta de Renomeação de Registradores em Arquiteturas Super Escalares

### C.1 Descrição da Estrutura

Nossa proposta utiliza uma arquitetura similar à descrita no capítulo 3 desta tese, mas eliminado-se o *future file* adicional. Neste caso, um único banco de registradores guarda tanto os registradores arquiteturais como os registradores de renomeação.

Se atrasarmos a alocação dos registradores físicos para estágios posteriores do *pipeline*, é possível obter uma redução significativa no número de registradores necessários para a renomeação.

Na arquitetura que estaremos investigando nesta seção, o banco de registradores possui múltiplas portas de leitura e de escrita. O *tag array* da nosso modelo anterior é substituído por uma tabela de mapeamento, separada do banco de registradores.

Essa tabela possui a seguinte estrutura: é indexada pelo número do registrador arquitetural desejado, obtendo-se na saída o valor do *tag* que identifica a instrução que vai produzir o resultado; **ou** o endereço do registrador físico que vai conter o resultado mais **recente** produzido para aquele registrador arquitetural. Um *bit* de validade indica se o endereço do registrador físico está válido ou não. Possui estrutura idêntica à tabela GMT proposta por González [72], conforme mostrado na Figura C.1.

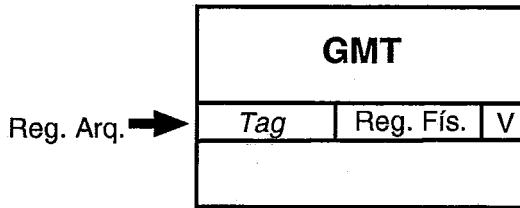


Figura C.1: Tabela de Mapeamento

Utilizamos também um *reorder buffer* com a estrutura mostrada na Figura C.2. O *reorder buffer* é muito simples e semelhante à proposta de [86]: nele são armazenados o código das instruções, o *tag* associado à instrução, o endereço do registrador físico alocado, um bit de “pronto” e, eventualmente, o resultado da instrução ou a informação de exceção ou predição incorreta de uma instrução desvio.

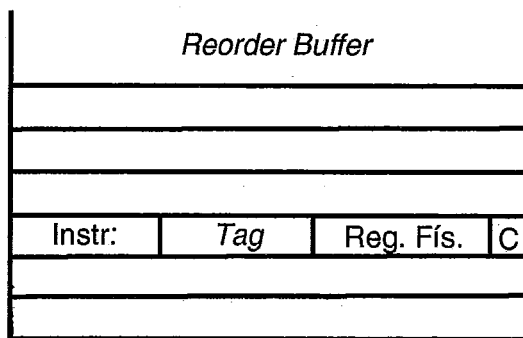


Figura C.2: Esquema Simplificado do *Reorder Buffer*

O armazenamento do resultado da operação foi uma adição necessária para evitar *deadlock* por ocasião da alocação dos registradores físicos, conforme será detalhado posteriormente.

Na saída do *reorder buffer* existe uma tabela que armazena o mapeamento “em-ordem” dos registradores arquiteturais para os registradores físicos. Este mapeamento é atualizado com o valor do registrador físico armazenado para as instruções que são concluídas a cada ciclo.

A Figura C.3 a seguir ilustra a organização da tabela de mapeamento, banco de registradores e um *reorder buffer*.

Nossa proposta permite reduzir o tempo entre a alocação de um registrador físico e sua liberação para reutilização, diminuindo assim o tamanho do banco de

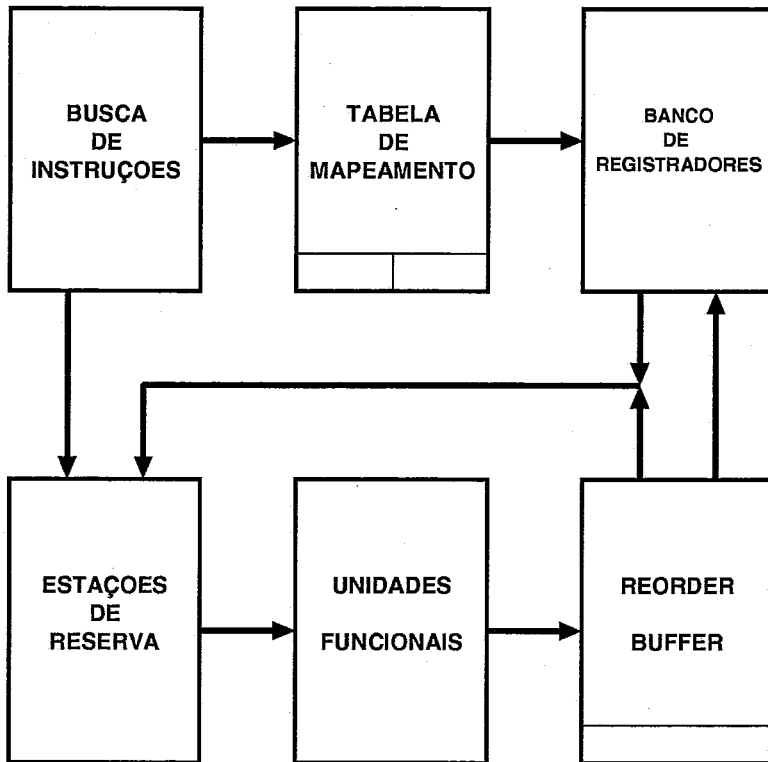


Figura C.3: Organização da Estrutura de Renomeação

registradores.

## C.2 Descrição do Funcionamento

No estágio de despacho os operandos das instruções são renomeados. Os operandos de saída são renomeados com o *tag* da estação de reserva para onde a instrução será enviada. Este valor de *tag* será copiado para a tabela de mapeamento na posição correspondente ao registrador arquitetural e o *bit* de validade será desligado.

As instruções sucessoras, que estiverem sendo despachadas neste ou em ciclos posteriores, ao endereçarem a tabela de mapeamento, receberão este *tag* no lugar dos operandos de leitura. Essas instruções aguardam nas estações de reserva até que um resultado com o *tag* correspondente seja produzido.

Note que até este momento não foi necessária a alocação de nenhum registrador físico. Esta alocação será feita quando a instrução estiver no último ciclo do estágio de execução, a partir de uma lista de registradores físicos livres.

Quando a execução da instrução terminar, o *tag*, o endereço do registrador físico



alocado e o resultado gerado são propagados para o banco de registradores, tabela de mapeamento e estações de reserva.

Cada entrada na tabela de mapeamento irá fazer uma comparação com o *tag* recebido. Em caso de coincidência, o identificador do registrador físico será copiado no campo correspondente e o *bit* de validade será ativado. As estações de reserva fazem a comparação do *tag* enviado com o *tag* armazenado para seus operandos e, no caso de coincidência, o resultado será copiado para a estação de reserva.

Em nosso modelo, **todos** os resultados são escritos no banco de registradores, independentemente de qualquer valor de *tag*. No algoritmo de Tomasulo, os resultados da instruções somente serão atualizados no banco de registradores caso o *tag* armazenado coincida com o *tag* transmitido.

O identificador do registrador físico armazenado na tabela de mapeamento será fornecido para novas instruções que forem despachadas. Quando as instruções forem copiadas para as estações de reserva, o conteúdo do registrador correspondente será copiado para as estações de reserva.

Resumindo, a atualização da tabela de mapeamento é feita nos seguintes casos:

- Com o *tag* da estação de reserva que vai gerar o resultado mais recente para aquele registrador arquitetural;
- Com o endereço do registrador físico como o resultado mais recente, caso o *tag* do resultado seja coincidente com o *tag* armazenado. Neste caso um bit de validade é setado, indicando que o campo está válido;
- Quando houver necessidade de recuperar o contexto, com o mapeamento “em-ordem” que é mantido atualizado na saída do *reorder buffer*.

Quando uma instrução é despachada para a estação de reserva, ela é colocada na primeira posição do *reorder buffer*, junto com o valor do *tag* da estação de reserva para a qual foi distribuída. O *bit* de pronto no *reorder buffer* é desligado. Se o *reorder buffer* estiver cheio, as instruções não podem mais ser despachadas.

Quando a instrução estiver no último ciclo de execução, será feita uma busca na lista de “livres”, para determinar qual registrador físico será alocado para o resultado desta instrução. Ao término da execução, o endereço do registrador físico alocado

será escrito na posição correspondente no *reorder buffer*, juntamente com a ativação do bit de “pronto”.

Quando a instrução que estiver no topo do *reorder buffer* for completada, ela será removida do *reorder buffer*. O identificador do registrador físico, que ocupa no mapa “em-ordem” a posição correspondente ao registrador arquitetural de saída da instrução, é removido e colocado na lista de “livres”.

No seu lugar será colocado o identificador do registrador físico salvo junto com a instrução que está sendo removida do topo do *reorder buffer*. Se a instrução no topo do *reorder buffer* for um desvio predito incorretamente, então todos os registradores físicos que não estiverem no mapa “em-ordem” serão liberados. O mapa “em-ordem” será copiado para o mapa atual. O comportamento para uma interrupção é idêntico ao comportamento para o desvio predito incorretamente.

Os resultados especulativos são mantidos no banco de registradores, enquanto houver possibilidade de serem reutilizados. São as seguintes as condições para que um registrador físico alocado possa ser liberado:

- O resultado da instrução foi gerado e escrito no registrador físico;
- Todas as instruções que foram despachadas para as estações de reserva, que necessitam deste resultado (*tag*) já o receberam;
- A instrução que gerou o resultado armazenado no registrador chega ao topo do *reorder buffer*, quando então é promovido a registrador arquitetural;
- Quando um outro registrador físico for promovido para registrador arquitetural, este registrador físico passa então para a lista de livres.

### C.3 Evitando o *Deadlock*

Entretanto, assim como outras propostas que atrasam a alocação dos registradores físicos [72, 51], procedimentos especiais são requisitados caso não haja registradores físicos disponíveis ao término da execução de uma instrução. Esses procedimentos, necessários para garantir o correto funcionamento do processador, são descritos a seguir.

Caso não haja registradores físicos disponíveis ao término da execução das instruções é possível que esta seja a instrução mais velha no *reorder buffer*. Neste caso, as demais instruções ficam impedidas de concluir e nenhum registrador físico será liberado, caracterizando um *deadlock*.

A solução proposta aqui é que o resultado desta instrução seja escrito no *reorder buffer*. Não haverá endereço do registrador físico para ser fornecido para a tabela de mapeamento. Ela continuará com o campo de *tag* inalterado, e este valor é que será fornecido para as instruções subseqüentes até que o registrador seja novamente renomeado.

Ao atingir o topo do *reorder buffer*, o registrador físico atualmente reservado para o registrador arquitetural será atualizado com o resultado da instrução que está armazenado no *reorder buffer*, sendo mantido na lista de “ocupados”. O valor do resultado e o *tag* da instrução que chegam ao topo do *reorder buffer* serão transmitidos para eventuais instruções que estiverem aguardando nas estações de reserva. O campo correspondente na tabela de mapeamento será atualizado se houver coincidência entre os *tags*.

Nesta situação o número de instruções que podem ser executadas em paralelo diminui, porque necessitam aguardar que as instruções cheguem ao topo do *reorder buffer*. Mas este atraso é razoável, dado que o resultado já foi transmitido para as instruções que aguardavam nas estações de reserva e somente novas instruções que venham a ser despachadas sofrerão esta restrição. Se considerarmos um esquema convencional, o despacho é suspenso quando os registradores físicos disponíveis para renomeação se esgotam.

## C.4 Avaliação

Nesta seção comparamos nossa proposta com os trabalhos publicados por González et al. [72, 73]. Estes trabalhos foram apresentados na seção 2.7 e, em nosso ponto de vista, apresentam as seguintes deficiências:

- Gera tráfego adicional pois o *tag* virtual alocado para a instrução que teve o seu registrador físico tomado tem que ser propagado para todas as instruções na janela de instruções para que o respectivo operando seja marcado como “não”

pronto. Como consequência, o número de caminhos de dado e comparadores na janela de instruções é aumentado;

- Não apenas as instruções que tiveram seus registradores físicos tomados, mas todas as instruções na cadeia de dependência que estiverem na janela de instruções devem ser re-executadas. Os valores médios são cerca de 10% para as instruções inteiras e 60% para as de ponto flutuante, segundo os próprios autores. Isto vai gerar principalmente uma maior pressão pelo uso das unidades funcionais e banco de registradores (mais registradores sendo lidos a cada ciclo);
- A alegação de que a re-execução é proveitosa é equivocada, já que a vantagem provém da execução antecipada das instruções, em relação a uma arquitetura com renomeação convencional. Em nosso modelo, esta mesma antecipação é conseguida sem custo associado à re-execução;

A proposta aqui colocada oferece algumas vantagens, que podemos resumir a seguir:

- Como no esquema de González, o endereçamento do banco de registradores é feito em duas etapas: primeiro a tabela de mapeamento para obter o endereço físico e no ciclo seguinte o acesso ao banco de registradores. Com o acesso dividido em dois ciclos, não há impacto sobre o tempo de acesso aos operandos;
- A tabela PMT apresentada por González responsável pela inclusão de um estágio extra no *pipeline*, não é necessária em nosso esquema. González faz a utilização da PMT, com busca associativa, para converter de *tag* virtual para endereço físico, quando a instrução chega ao topo do *reorder buffer*. Em nosso esquema, o endereço físico é colocado no *reorder buffer* junto com o resultado produzido pela instrução;
- Em relação ao esquema convencional de renomeação, a nossa proposta utiliza um menor número de registradores físicos, por haver um menor tempo entre a alocação e a liberação destes registradores;
- A recuperação de contexto é bem mais simples em nosso esquema. Na proposta de González todas as instruções no *reorder buffer* entre o ponto que ocorreu

a exceção ou predição incorreta e a instrução mais recente, tem que ser examinadas e o mapeamento desfeito uma a uma. Em nossa proposta basta copiar o mapeamento “em-ordem” para a tabela de mapeamento;

- O número de portas de leitura do banco de registradores é o mesmo que no esquema convencional de Tomasulo, já que os *tags* dos operandos são fornecidos pela tabela de mapeamento. Os valores serão fornecidos pelo banco de registradores apenas se o registrador arquitetural correspondente não tiver sido renomeado;
- O número de portas da tabela de mapeamento é igual ao do *tag array* da proposta de nossa tese, já que os procedimentos são os mesmos, substituindo-se o resultado da instrução pelo identificador do registrador físico alocado.
- O esquema proposto para evitar o *deadlock* necessita de um número de registradores no *reorder buffer* deve igual à capacidade do *reorder buffer* menos o número de registradores disponíveis para renomeação. Por exemplo, se o tamanho do *reorder buffer* for de 32 entradas, com 48 registradores físicos e 32 arquiteturais, então o número máximo de registradores necessários no *reorder buffer* será de  $32 - (48 - 32) = 16$  registradores.
- Mecanismos de reuso [87] podem ser aplicados também ao nosso esquema, embora não façamos aqui nenhuma análise aprofundada desse tema.

## C.5 Resumo

O trabalho de González [72] indica que uma arquitetura com 48 registradores com alocação dos registradores no estágio de execução é equivalente a um esquema convencional de renomeação com 64 registradores. Para os programas de ponto flutuante, seriam necessários 77 registradores contra 101 registradores de uma organização convencional [73].

Acreditamos que os mesmos resultados possam ser obtidos no esquema proposto, pois este também realiza a alocação de registradores físicos durante o estágio de execução *pipeline*.

A nossa opção por armazenar os resultados no *reorder buffer*, caso não haja registradores físicos disponíveis, implica em um gasto adicional de registradores. Entretanto, esses registradores não estão no caminho crítico e podem ter um tempo de acesso menos restritivo (em dois ciclos, por exemplo) do que os do banco de registradores. Além disto, nenhuma busca associativa é necessária para o acesso aos dados desses registradores, o que vem a simplificar o *hardware* utilizado.

Outras opções estão sendo estudadas, como por exemplo armazenar os resultados das instruções também nas estações de reserva, o que, além de evitar o *deadlock*, poderia servir para um esquema de reutilização de valores [19, 87].