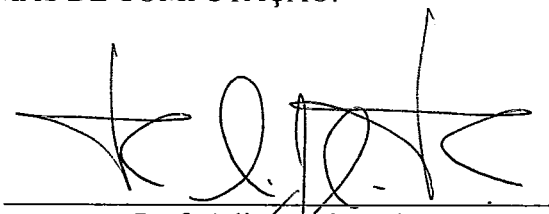
MEMORIZAÇÃO DINÂMICA DE *TRACES* COM REUSO DE VALORES DE INSTRUÇÕES DE ACESSO À MEMÓRIA

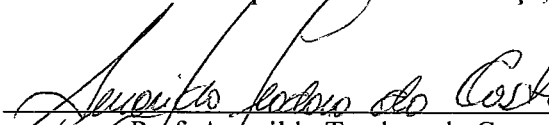
Luiz Marcio Faria de Aquino Viana

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 MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS DE COMPUTAÇÃO.

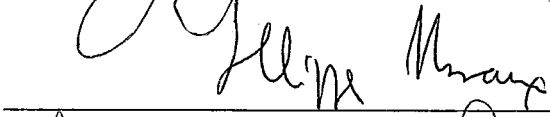
Aprovada por:



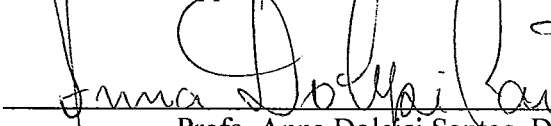
Prof. Felipe Maia Galvão França, Ph.D.



Prof. Amarildo Teodoro da Costa, D.Sc.



Prof. Philippe Olivier Alexandre Navaux, Dr. Ing.



Profa. Anna Dolejsi Santos, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2002

VIANA, LUIZ MARCIO FARIA DE AQUINO

Memorização Dinâmica de *Traces* com Reuso de Valores de Instruções de Acesso à Memória [Rio de Janeiro] 2002

XII, 118 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2002)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Reuso Dinâmico de *Traces*

2. Arquitetura de Processador

I. COPPE/UFRJ II. Título (série)

DEDICATÓRIA

À Mila, minha esposa, e aos meus filhos, Luiz Felipe e Maria Julia,
por toda a alegria e força que me proporcionam.

AGRADECIMENTOS

Agredeço a Deus pela força e luz recebida durante o desenvolvimento deste trabalho e pela oportunidade que eu tive em ampliar o meu conhecimento.

Aos meus orientadores, Eliseu Monteiro Chaves Filho e Felipe Maia Galvão França pelos valiosos conhecimentos recebidos e pela orientação dedicada e profissional que foi fundamental para o amadurecimento e finalização deste trabalho.

Agradeço em especial ao apoio recebido do amigo Amarildo Teodoro da Costa pelas diversas conversas e idéias que trocamos durante o desenvolvimento desta pesquisa e que foram fundamentais para a compreenção do mecanismo base deste trabalho.

À todos os companheiros de trabalho da COPPE/Sistemas que de forma direta ou indireta contribuiram com o desenvolvimento deste trabalho e em especial aos companheiros de turma Álvaro da Silva Ferreira, Magnos Martinelli, Élcio Pinecchi, Alexis Braga Kropotoff e Igor Briglia Habib de Almeida Alves pelo incentivo mútuo na conquista de nossos objetivos.

À minha esposa, Mila, e aos meus filhos, Luiz Felipe e Maria Julia, pelo apoio, carinho e compreenção recebidos.

Aos meus pais, José Luiz de Aquino Viana e Maria Cristina Faria de Aquino Viana, pelo apoio e constante incentivo recebidos durante toda a minha vida.

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

MEMORIZAÇÃO DINÂMICA DE *TRACES* COM REUSO DE VALORES DE INSTRUÇÕES DE ACESSO À MEMÓRIA

Luiz Marcio Faria de Aquino Viana

Março/2002

Orientadores: Eliseu Chaves Filho
Felipe Maia Galvão França.

Programa: Engenharia de Sistemas e Computação

No passado, com o objetivo de aumentar o desempenho dos processadores, as novas tecnologias focavam a otimização da velocidade dos processadores sem a devida avaliação da quantidade de trabalho redundante executado. Recentemente, novos mecanismos foram introduzidos com o intuito de prover desempenho através da redução da quantidade de trabalho redundante executado por um processador e, entre estes, as técnicas de reuso dinâmico de *traces* vem se destacando bastante. Este trabalho introduz o novo mecanismo de reuso dinâmico de *traces* denominado *Dynamic Trace Memoization with Reuse of Memory Values*, DTM*m*, que estende as funcionalidades do mecanismo original, DTM (*Dynamic Trace Memoization*), com a adição de reuso de valores de instruções de acesso à memória. Os experimentos realizados com o DTM*m* apresentaram um crescimento médio de 4.7% na aceleração sobre o mecanismo original e 7.9% sobre a arquitetura base para um *benchmark* composto por um subconjunto de programas do SPECint' 95.

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

DYNAMIC TRACE MEMOIZATION WITH REUSE OF MEMORY ACCESS INSTRUCTIONS' VALUES

Luiz Marcio Faria de Aquino Viana

Março/2002

Advisors: Eliseu Chaves Filho
Felipe Maia Galvão França.

Department: Engenharia de Sistemas e Computação

In the past, with the objective to improve the speed of processors, the new technologies were looking for optimizations in processors speed without taking profit from redundant work. Recently, new mecanisms were introduced with the intention to improve speed with the reduction in the quantities of redundant work, and dynamic trace reuse techniques have been introduced. This work introduce a new scheme for trace reuse named Dynamic Trace Memoization with Reuse of Memory Values, DTM*m*, which extend the funtionality of DTM (Dynamic Trace Memoization), by adding reuse of values of memory access instructions. Experiments done with DTM*m* shows a growth of 4.7% in speed over the original mecanism and 7.9% over the superscalar base architecture for a benchmark based on a subset of programs from SPECint '95.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os recentes avanços obtidos na produção de circuitos integrados proporcionaram novos rumos na pesquisa e no desenvolvimento de mecanismos de *hardware* capazes de estender o poder de processamento dos computadores atuais. Assim, constantemente nos deparamos com novos mecanismos que quebram paradigmas no campo da arquitetura de computadores.

No passado, por falta de tecnologia apropriada para implementa-los, tais mecanismos eram freqüentemente abandonados ou demoravam muito tempo até que a industria os adotassem. Porém, com a evolução do processo de manufatura de *hardware* crescendo a taxas extremas, os novos mecanismos estão sendo absorvidos de forma cada vez mais rápida pela industria.

Os esforços contínuos no desenvolvimento de tecnologias que aumentam o desempenho dos processadores foram durante muito tempo dedicados exclusivamente à diminuição do ciclo de relógio e ao desenvolvimento de novas técnicas de exploração do paralelismo no nível de *hardware*. Porém quando falamos de desempenho de um processador nos referimos à sua capacidade de realizar uma determinada tarefa em um determinado tempo, e quando nos esforçamos em prover desempenho a um processador, desconsiderando o trabalho necessário ao cumprimento de uma tarefa, estamos deixando de avaliar a quantidade de trabalho redundante produzido pelo processador e que poderia ser evitado economizando tempo de processamento.

Pela equação de desempenho [15], apresentada a seguir (**Equação 1.1**), temos que o desempenho de um processador em realizar uma tarefa pode ser definido por:

$$\mathbf{CPUtime = IC \times CPI \times T}$$

Onde, **IC** é o número total de instruções executadas,
CPI é o número médio de ciclos por instrução, e
T é o tempo médio de um ciclo.

Equação 1.1: Equação de desempenho de um processador.

Analisando a equação de desempenho, **Equação 1.1**, observamos que pode-se melhorar o desempenho de um processador reduzindo o número de instruções necessárias ao cumprimento de uma determinada tarefa e para isto, precisamos de mecanismos que identifiquem e eliminem o trabalho redundante existente em um programa.

Atualmente novos mecanismos estão surgindo com o intuito de prover tal capacidade aos processadores modernos e o conceito de localidade de valores se tornou chave destes desenvolvimentos.

O conceito de localidade de valores foi inicialmente introduzido por [20] e expressa a tendência que os valores produzidos em uma unidade de armazenamento tem de se repetirem ao longo da execução de um programa. O motivo pelo qual isso ocorre se deve ao fato dos programas reais serem construídos para resolverem problemas genéricos e não para um conjunto limitado de entradas.

Existem vários mecanismos que exploram o conceito de localidade de valores e foram modelados a partir deste principio, porém existem apenas duas vertentes de pesquisa apoiadas nele. A primeira consiste de mecanismos de previsão de valores que procuram antecipar o resultado das instruções usando como referência o histórico de execuções anteriores, permitindo a execução especulativa de instruções com dependência

de dados verdadeira. Entre os mecanismos que seguem esta vertente podemos citar *last value prediction* [20,21], *stride prediction* [10] e *context based prediction* [31]. A segunda vertente procura eliminar a execução redundante através do reuso de instruções ou trechos de códigos dos programas, e como referência a esta linha de pesquisa podemos citar: *dynamic instruction reuse* [33], *block reuse* [16,17] e *trace reuse* [3,8,13].

Entre as linhas de pesquisa que exploram o conceito de localidade de valores os mecanismos que se baseiam no reuso de *traces* de instruções vem se destacando bastante pelos resultados obtidos, pelas possibilidades de desenvolvimento e pelos estudos que podem ser estendidos a partir deles.

1.1 MOTIVAÇÕES E OBJETIVOS

O objetivo deste trabalho é estender a funcionalidade de uma das técnicas de trace reuse denominada **DTM** [8], adicionando instruções de acesso à memória ao conjunto de instruções válidas do mecanismo e possibilitando desta forma que instruções que acessam memória possam fazer parte dos traces de instruções.

O motivo que levou ao desenvolvimento desta pesquisa, foi o fato do mecanismo **DTM** ser uma técnica nova e que produz ganhos significativos de desempenho. Porém, esta técnica não reusa valores de instruções de acesso à memória e conforme podemos ver na tabela (**Tabela 1.1**), este grupo de instruções representam 34% das instruções executadas por um programa e, além disso, conforme [31], instruções de acesso à memória possuem grande localidade de valores, sendo que aproximadamente 30% delas poderiam ser reusadas ao longo da execução de um programa. Assim a inclusão de instruções de acesso à memória ao **DTM** pode levar a ganhos consideráveis de desempenho.

Posição	Tipos de Instruções 80x86	% Instruções Executadas
1	Load	22 %
2	Conditional branch	20 %
3	Compare	16 %
4	Store	12 %
5	Add	8 %
6	And	6 %
7	Sub	5 %
8	Move reg-reg	4 %
9	Call	1 %
10	Return	1 %
	Total =	96 %

Tabela 1.1: Distribuição das instruções executadas no **SPECint '92**.

Para o desenvolvimento desta pesquisa foi construído um simulador de uma arquitetura superescalar executando o conjunto de instruções do processador **SPARC v7**, sendo que este simulador foi modelado segundo os conceitos de orientação a objetos onde os seus componentes podem ser estendidos, modificados ou removidos de forma bastante simples proporcionando grande flexibilidade à ferramenta.

Esta pesquisa também contribuiu para a avaliação dos resultados obtidos com o **DTM** em uma arquitetura diferente da originalmente estudada e usando outro conjunto de instruções. Além disso, nesta pesquisa é introduzido um novo esquema de formação e reuso de *traces* denominado **DTM*m***, que estende o modelo originalmente proposto em [8] com a inclusão de instruções de acesso à memória. Nos experimentos realizados com um subconjunto de programas do *benchmark* **SPECint '95**, o **DTM*m*** obteve aceleração média de 7.9% e média harmônica de 5.4% sobre a arquitetura base contra um resultado de 3.2% de média e 2.5% de média harmônica obtido pelo mecanismo original.

No capítulo à seguir, apresentaremos alguns conceitos iniciais e os mecanismos de previsão de valores e reuso de instruções que formam a base de conhecimento usada no desenvolvimento do **DTM*m***. O **Capítulo 3** apresenta o mecanismo original, **DTM**, e

alguns aspectos de sua implementação na arquitetura do simulador. No capitulo seguinte, **Capitulo 4**, este mecanismo é estendido com a inclusão de instruções de acesso à memória, e no **Capítulo 5** é apresentado o **DTM*m***, que é uma variação do mecanismo DTM original que separa o reuso de instruções de leitura de memória do mecanismo de reuso de *traces* para reduzir o custo de implementação do mecanismo. Em seguida, **Capítulo 6**, são apresentadas as análises dos resultados obtidos nos experimentos realizados e por último, **Capítulo 7**, temos a conclusão final do trabalho.

Capítulo 2

Fundamentos e Trabalhos Correlatos

Neste capítulo será apresentado o conceito de localidade de valores além de alguns mecanismos que exploram este conceito com a finalidade de eliminar ou atenuar os problemas provocados pela dependência de dados.

2.1 LOCALIDADE DE VALORES

Em [20] foi introduzido o conceito de localidade de valores. Este conceito expressa a tendência que um valor já visto possui de se repetir em uma unidade de armazenamento ao longo da execução de um programa.

As medidas realizadas em [20], demonstraram que os resultados produzidos por operações de leitura de memória exibem 50% de localidade de valor para um histórico que armazena o último resultado produzido por estas operações, e exibem 80% de localidade de valor para um histórico que armazena os dezesseis últimos resultados obtidos pela execução das instruções.

Os gráficos da **Figura 2.1** a seguir, apresentam os resultados obtidos em [20] para um conjunto de programas dos *benchmarks* de inteiro **SPECint '92, SPECint '95,** algumas aplicações de processamento de imagem e utilitários de uso geral, além do uso de quatro programas do *benchmark* de ponto flutuante do **SPECfp '92,** executados por simuladores das plataformas **Alpha AXP** e **PowerPC**.

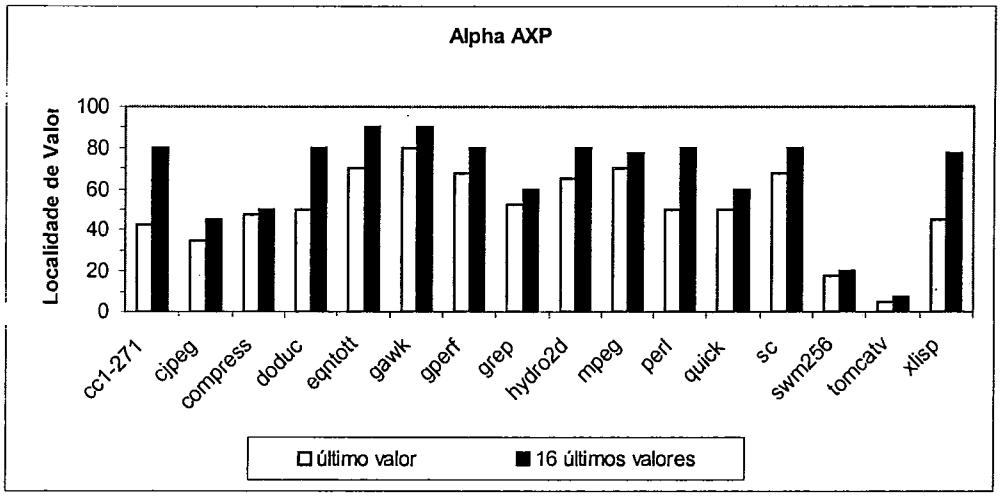


Figura 2.1a: Localidade de valores encontrada no simulador **Alpha AXP**.

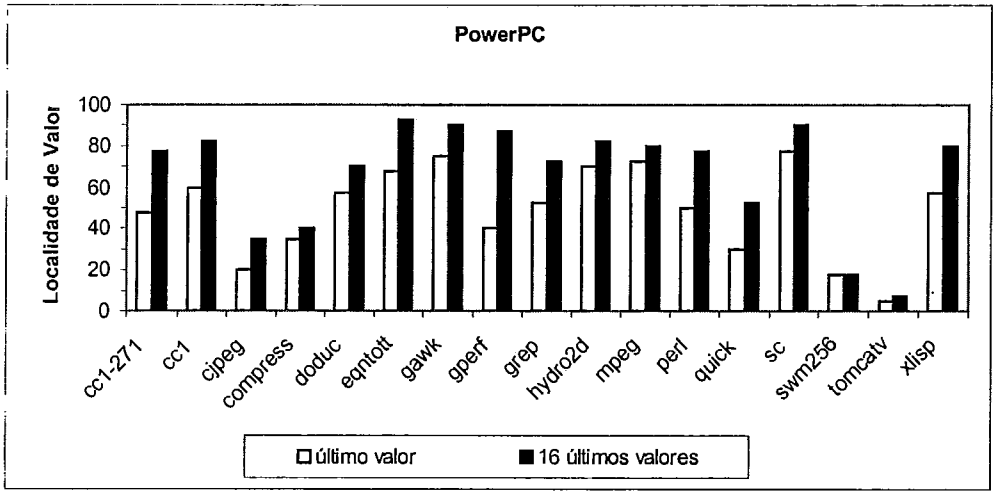


Figura 2.1b: Localidade de valores encontrada no simulador **PowerPC**.

O motivo pelo qual a localidade de valor existe, se deve a natureza genérica dos programas reais que são construídos para resolverem problemas para conjuntos de entradas distintos e desta forma estes programas são construídos tendo como base *loops* e sub-rotinas que freqüentemente manipulam os mesmos dados de entrada.

Por exemplo, programas que possuem poucas variações nos conjuntos de entradas, tais como processadores de textos e planilhas eletrônicas executam com freqüência processamento sobre os mesmos dados. Também a utilização de constantes de

dados que freqüentemente são armazenadas em memória e usadas na fase inicial de uma computação favorecem a localidade de valor.

Mecanismos de predição de desvios exploram o conceito de localidade de valores [14,18,22,36], porém em [20], este conceito foi introduzido em um experimento com valores que endereçam 32 e 64 *bits* de dados, isto é, para um domínio de valores muito maior, confirmando a previsibilidade de valores de dados. A partir deste conceito vários mecanismos foram criados com o intuito de aumentar o desempenho dos processadores e entre eles podemos citar os mecanismos de previsão de valores e os mecanismos de reuso de valores que serão apresentados neste capítulo.

2.2 TÉCNICAS DE PREVISÃO DE VALORES

As técnicas de previsão de valores procuram antecipar o resultado de uma instrução antes da sua execução atenuando os efeitos da dependência de dados. Estes mecanismos se baseiam em uma arquitetura onde uma tabela de classificação de instruções denominada *classification table*, **CT**, é usada para identificar as instruções que possuem melhores condições de terem seus resultados previstos corretamente, além de identificar qual o tipo de preditor mais adequado para a previsão do resultado da instrução, quando são assumidos múltiplos preditores.

Estas técnicas por serem de natureza especulativa, exigem que o resultado previsto para uma instrução, seja validado após a execução dela. Assim se o resultado for previsto incorretamente o processador sofrerá penalidades, pois terá que executar as instruções novamente usando o valor correto. Deste modo, o uso de uma tabela de classificação das instruções permite selecionar as instruções com maior chance de acerto na previsão minimizando as penalidades impostas.

As tabelas de classificação são freqüentemente construídas usando métodos de predição do tipo contador saturado para avaliar as chances de previsão correta dos valores das instruções. Após determinar se o resultado de uma instrução pode ou não ser previsto

corretamente uma segunda tabela denominada *value prediction table*, **VPT**, é acessada com o intuito de obter o valor previsto.

Após o acesso a **VPT**, o resultado previsto fica disponível para as instruções seguintes que requerem este valor, e a instrução entra em execução para que o resultado possa ser obtido e posteriormente comparado com o resultado previsto. Caso haja discordância as instruções executadas especulativamente são escalonadas novamente para execução com o valor correto.

Em [20] a tabela de predição de valores é denominada *load value prediction table*, **LVPT**, e neste mecanismo uma unidade adicional é usada na validação dos resultados previstos. Esta unidade recebe o nome de *constant verification unit* ou **CVU** e serve para verificação de instruções de leitura da memória que possuem elevada taxa de acerto nas previsões e são classificadas como constantes pela tabela de classificação. A **CVU** opera como uma memória totalmente associativa que mantém coerência com os valores correspondentes na memória *cache* e desta forma o seu emprego evita que um acesso à memória seja efetuado para obtenção do resultado de operações classificadas como constantes.

2.2.1 IMPLEMENTAÇÃO TÍPICA DE PREVISÃO DE VALORES

A **Figura 2.2**, apresenta o diagrama de blocos de um mecanismo de previsão de valores típico [21]. Nesta figura observamos que as tabelas de classificação, **CT**, e de valores previstos, **VPT**, são acessadas no estágio de busca usando o endereço da instrução como referência. Para possibilitar que múltiplos resultados de uma mesma instrução estejam presentes na **VPT**, um índice pode ser concatenado ao endereço da instrução e usado como referência em uma tabela associativa por conjunto. Neste caso o uso de um preditor que possa identificar eficientemente o valor mais provável entre os valores disponíveis nesta tabela se torna fundamental para o mecanismo.

No momento do despacho da instrução, o resultado da tabela de classificação é consultado e se ele indicar que a instrução é previsível, o valor previsto será antecipado às instruções seguintes.

Após a execução da instrução, um estágio de validação adicionado pelo mecanismo é usado para verificação do resultado previsto e se o resultado obtido pela execução da instrução não corresponder ao valor previsto às instruções seguintes são escalonadas novamente.

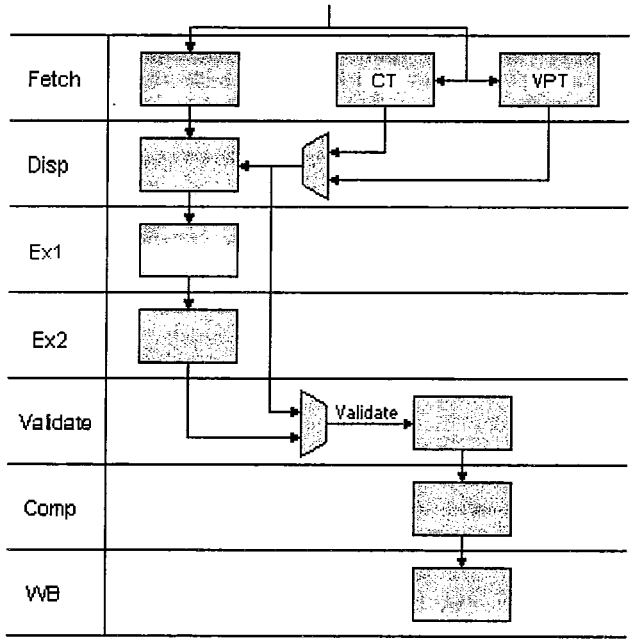


Figura 2.2: Diagrama de blocos do mecanismo de previsão de valores.

2.2.2 ANALISANDO DIFERENTES TIPOS DE PREDITORES

As técnicas de previsão de valores são bastante sensíveis à qualidade da tabela de classificação e ao tipo de preditor usado. Podemos classificar a seqüência de valores produzidos por uma instrução como constante, seqüencial ou não-seqüencial [31], conforme apresentado na **listagem 2.1**.

Constante	2, 2, 2, 2, ..., 2
Seqüencial	1, 3, 5, 7, ..., 2n+1
Não-seqüencial	1,15, 20, 14, ..., ?

Listagem 2.1: Tipos de valores produzidos por uma instrução.

Sendo que as seqüências de valores **seqüencial** e **nao-seqüencial** ainda podem ter ou não repetição conforme apresentado na **listagem 2.2** a seguir.

Seqüencial com repetição	1, 3, 5, 1, 3, 5, ...
Não-seqüencial com repetição	1, 15, 20, 14, 1, 15, 20, 14, ...

Listagem 2.2: Tipos de valores com repetições produzidos por uma instrução.

Tomando como base às classificações dos tipos de seqüências de dados que podem ser produzidas por uma instrução, podemos definir dois tipos básicos de preditores: Preditor computacional que efetua previsões com base na avaliação de expressões sobre valores previamente obtidos e preditor com base no contexto dos dados que aprende como a seqüência de dados se comporta e procura reproduzir os valores futuros com base nas seqüências anteriores.

Entre os preditores do tipo computacional podemos destacar *Last Value Prediction* de [20,21], e *Stride Prediction* apresentado em [10]. Já entre os preditores com base no contexto da seqüência de dados temos o *Context Based Prediction* introduzido em [31].

Last Value Prediction

Este é o modelo tradicional de preditor inicialmente apresentado em [20] para previsão de valores de operações de leitura da memória, e foi em seguida estendido para todas as instruções em [21]. Neste modelo é armazenado o valor do último resultado produzido por uma instrução, e se na ocorrência da instrução ela estiver classificada como previsível este valor é usado como valor previsto.

Stride Prediction

Este modelo de preditor foi inicialmente introduzido por [10] e efetua a computação do valor previsto somando o último valor retornado pela instrução com a diferença entre ele e o penúltimo valor produzido. Para efeito de implementação são armazenados o valor atual e o próximo valor da instrução evitando a necessidade de computação deste valor no momento da utilização.

Este tipo de preditor consegue prever com eficiência instruções que retornam valores constantes ou que possuem comportamento repetitivo constante, como ocorre com o preditor anterior, porém este método prevê também seqüências de valores como as produzidas por contadores.

Context Based Prediction

O modelo de preditor com base no contexto das seqüências de dados, procura aprender o comportamento dos valores produzidos por uma instrução aplicando um modelo de contexto finito ou **FCM** [31]. A figura a seguir, **Figura 2.3**, apresenta alguns modelos de contexto finito com profundidade variando de um a quatro valores de seqüência.

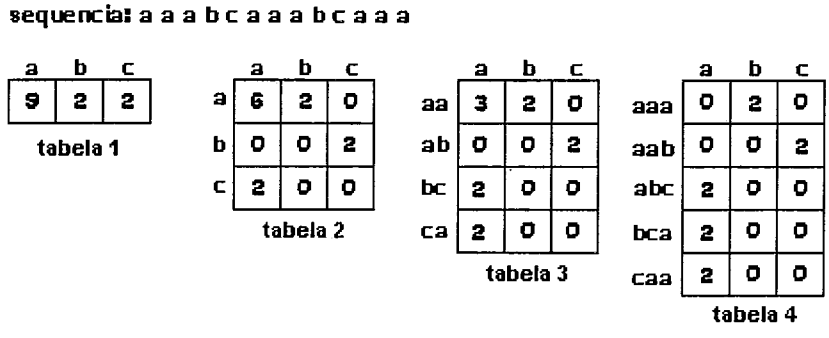


Figura 2.3: Modelos de contexto finito de profundidade de 1-4 valores.

Para a seqüência de valores apresentado na **Figura 2.3** existe grande chance do próximo valor da seqüência ser o valor **a** se usarmos como base a **tabela 1**, pois a freqüência deste valor é maior dentro da tabela. Para um histórico com os dois últimos valores produzidos pela instrução se o valor atual é **a** a chance do próximo valor ser **a** é maior do que a chance deste valor ser **b** ou **c**, conforme apresenta a **tabela 2.**

Na **tabela 3**, temos um histórico de três, onde dado que os dois últimos valores foram **a** e **a** o próximo valor da seqüência será provavelmente **a**, e na **tabela 4** temos um histórico de quatro onde os três últimos resultados foram **a, a** e **a**, assim o próximo valor deverá ser **b**, pois a incidência deste valor após a ocorrência de três valores **a** é maior.

Conforme podemos observar, este modelo procura identificar dentro do contexto do programa qual o comportamento da seqüência de valores gerada pela instrução, sendo este modelo bastante eficiente na previsão de resultados constantes, de seqüências repetitivas finitas e de estruturas periódicas não seqüenciais. Porém a quantidade de valores que precisam ser armazenados para formar um histórico de resultados que apóie eficientemente este modelo inibe o seu uso.

Experimentos conduzidos em [31] usando o *benchmark* de inteiros **SPECint '95** demonstram ser este o modelo mais eficiente de predição de valores. Os resultados obtidos nestes experimentos estão reproduzidos no gráfico da **Figura 2.4**, onde **L** representa *last value prediction*, **S** significa *stride prediction* e, **FCM1**, **FCM2** e **FCM3** são preditores do tipo *context based prediction* com histórico de uma, duas e três execuções respectivamente.

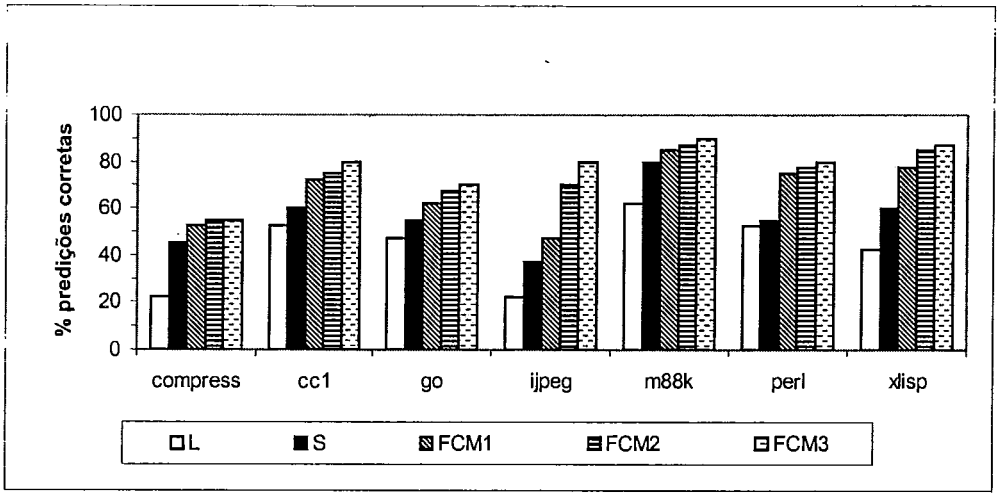


Figura 2.4: Taxa de acerto nas previsões por tipo de preditor.

Os experimentos conduzidos em [31], também mostraram que o uso de um modelo híbrido composto por *Stride Prediction* e *Context Based Prediction*, pode proporcionar um eficiente desempenho a um custo menor de implementação do que usando apenas o mecanismo com base no contexto da seqüência de dados.

Nas próximas seções serão apresentados alguns mecanismos que usam técnicas de reuso de valores. Estes mecanismos formam a base de implementação do **DTM*m***.

2.3 REUSO DINÂMICO DE INSTRUÇÕES SIMPLES

Em [33] foi introduzida à técnica de reuso dinâmico de instrução, que diferente das técnicas de previsão, exploram o conceito de localidade de valores através do reuso dos resultados das operações.

Os experimentos realizados em [33] apresentaram três técnicas de reuso dinâmico de instruções denominadas **Sv**, **Sn** e **Sn+d**. Estes três esquemas tomam como base uma estrutura comum formada por uma tabela que armazena os resultados produzidos por instruções já finalizadas e reusa estes valores quando as instruções correspondentes são

novamente executadas com os mesmos valores de entrada. A esta estrutura comum foi dado o nome de *reuse buffer*, **RB**.

A **Figura 2.5** apresenta a estrutura do *reuse buffer* que é uma tabela usada para armazenamento dos resultados das instruções. Esta tabela é indexada pelo endereço da instrução e dispõem de um mecanismo capaz de efetuar invalidações seletivas das entradas na ocorrência de determinados eventos.

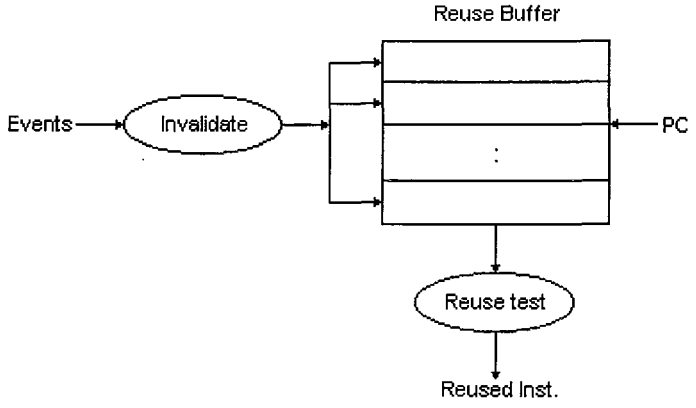


Figura 2.5: Estrutura genérica do *reuse buffer*.

Cada entrada na tabela do *reuse buffer*, **Figura 2.6**, é formada por um campo **tag** de identificação cujo valor é obtido a partir do endereço da instrução, pelos campos **operand1** e **operand2** responsáveis por armazenar informações dos operandos de entrada, por um campo **res** que armazena o resultado da instrução, e pelo endereço de acesso à memória das instruções de leitura armazenado no campo **addr**, além de *bits* de sinalização usados para indicar se o valor armazenado no campo **res** é ou não válido para reuso.

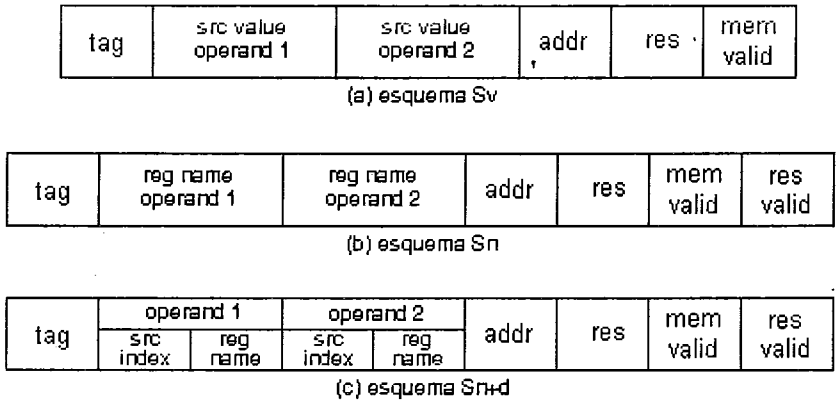


Figura 2.6: Estrutura das entradas do *reuse buffer*.

Em [33], três esquemas de reuso dinâmico de instruções foram apresentados, o primeiro **Sv** efetua o reuso de instruções tomando como base os valores dos operandos de entrada, o segundo esquema denominado **Sn** processa o reuso de instruções através do registro dos operandos fonte dos dados e não mais com base nos valores de entrada. Enquanto o terceiro esquema, **Sn+d**, estende os anteriores adicionando a capacidade de reconhecimento de seqüências de instruções dependentes.

2.3.1 ESQUEMAS DE REUSO DINÂMICO DE INSTRUÇÕES

Esquema Sv

O esquema **Sv** é a implementação imediata do reuso de instruções, onde os valores dos operandos de entradas são adicionados às entradas correspondentes no *reuse buffer* e usados na verificação do reuso.

Conforme é apresentado na **Figura 2.6a**, cada entrada do *reuse buffer* armazena em **operand1** e **operand2** o valor dos operandos de entrada das instruções, e um *bit* de sinalização, **mem valid**, que indica se o resultado pode ou não ser usado por uma instrução de leitura da memória.

Quando uma instrução vai ser despachada é feita a comparação dos valores dos operandos de entrada com os respectivos valores armazenados na entrada do *reuse buffer* e se os valores forem iguais ocorrerá o reuso. Se a instrução for de leitura de memória além da verificação dos valores dos operandos de entrada será necessário consultar também o *bit* **mem valid** para confirmar se o resultado da operação de memória pode ou não ser reusado. Caso o resultado da operação não possa ser reusado será reusado apenas o endereço do acesso à memória.

Se no momento do despacho os valores dos operandos de entrada da instrução não estiverem disponíveis o teste de reuso não será efetuado e a instrução será executada normalmente. Instruções de escrita na memória fazem reuso apenas do endereço de acesso à memória e não do resultado a ser armazenado.

Invalidações das entradas do *reuse buffer* ocorrem quando uma operação de escrita à memória é encontrada. Neste ponto as entradas do *reuse buffer* são consultadas à procura de entradas contendo instruções de leitura que lêem do mesmo endereço de memória que está sendo modificado, e nas entradas encontradas o sinalizador **mem valid** é anulado invalidando os resultados das operações de leitura.

Observe que este esquema invalida de forma associativa apenas instruções de leitura da memória, além disso, este esquema pode ser implementado com o uso de duas tabelas, uma para reuso do resultado das instruções e do endereço de acesso à memória das operações de leitura e escrita, e outra para reuso do resultado de operações de leitura da memória. Está organização evita o uso de um mecanismo de invalidação seletiva no *reuse buffer* mantendo este recurso em uma segunda tabela que pode possuir menos entradas.

Esquema Sn

Neste esquema cada entrada do *reuse buffer*, **Figura 2.6b**, armazena os identificadores dos operandos fontes da instrução e não os valores deles. Assim, o teste

de reuso tem como base à verificação de que os registradores fontes não sofreram modificações desde a última execução da instrução. Para monitorar e identificar as entradas com resultados válidos para reuso foi adicionado um campo **result valid** que sinaliza se o valor armazenado é valido.

Quando uma instrução vai ser despachada é verificado se existe alguma entrada no *reuse buffer* que corresponde a uma instância de execução anterior. Se existe uma instância correspondente, o campo **result valid** é consultado para verificar se o valor armazenado na tabela é valido para reuso.

Para instruções de acesso à memória o mecanismo consulta o campo **result valid** para verificar se o endereço da operação de acesso à memória pode ser reusado. Para operações de leitura da memória é analisado também o campo **mem valid** para verificar se o valor da instrução pode ou não ser reusado.

Neste esquema as invalidações ocorrem por dois motivos. Primeiro, quando uma operação modifica o valor contido em um registrador usado como operando fonte de alguma entrada do *reuse buffer*, neste caso o campo **result valid** é invalidado, e quando uma instrução de escrita modifica o valor armazenado em um endereço de memória referenciado por alguma instrução presente na tabela de reuso, onde o campo **mem valid** é invalidado.

Esquema Sn+d

O esquema **Sn+d** é similar ao esquema **Sn** e usa os identificadores dos operandos fontes e não os valores para verificação de reuso, porém este esquema procura identificar a relação de dependência entre as instruções para evitar a invalidação excessiva das entradas do *reuse buffer*, comum no esquema anterior, **Sn**.

Neste esquema as entradas do *reuse buffer* possuem estrutura similar à encontrada no esquema **Sn**, porém cada operando de entrada possui também um índice, **src index**,

que aponta para a instrução fonte do dado usado pelo registrador e que está localizada no *reuse* buffer, como pode ser visto na **Figura 2.6c**. A adição do campo **src index** fornece um método eficiente para verificar se instruções dependentes podem ser reusadas em um mesmo ciclo.

O esquema **Sv** não implementa um mecanismo que identifica dependências e na presença de instruções dependentes este mecanismo não pode verificar a redundância destas instruções em um mesmo ciclo. O esquema **Sn** efetua invalidações na presença de instruções dependentes, pois o armazenamento das instruções para reuso leva em consideração os identificadores dos registradores fontes e não seus valores. Deste modo, qualquer instrução executada posteriormente e que escreve nos operandos fontes invalidam as entradas correspondentes no *reuse buffer*.

No esquema **Sn+d** o reuso de instruções dependentes presentes na janela de despacho em um mesmo ciclo é resolvido com o uso de ponteiros para as entradas correspondentes das instruções fontes de seus operandos que também estão no *reuse buffer*. Deste modo, se os registradores fontes da instrução dependente forem modificados por alguma instrução posterior, esta instrução não será invalidada, pois os valores de seus operandos são produzidos por uma instrução presente no *reuse buffer*, e se as instruções fontes forem reusadas a instrução dependente será também reusada.

Para auxiliar na identificação de dependência entre instruções e no teste de reuso de cadeias de dependência em um mesmo ciclo foi adicionado ao mecanismo base uma tabela auxiliar denominada *register source table*, **Figura 2.7**, que possui uma entrada correspondente a cada registrador da arquitetura que armazena nestas entradas o índice da instrução presente no *reuse buffer* e que modificou por último o registrador.

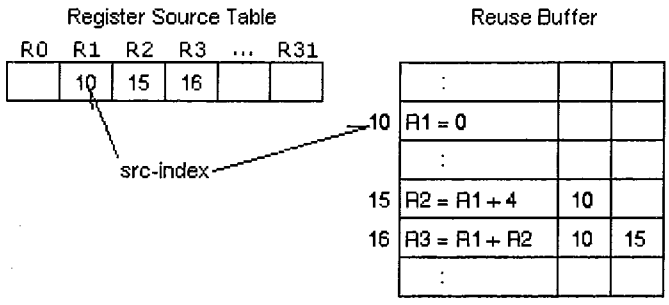


Figura 2.7: *Register source table* armazena índices do *reuse buffer.*

A verificação de reuso no esquema **Sn+d** é similar ao teste de reuso realizado pelo esquema **Sn**, onde no momento do despacho o endereço da instrução é usado na obtenção da entrada correspondente no *reuse buffer.* Existindo uma entrada correspondente é realizada a verificação do sinalizador **res valid**, que indica se o resultado da instrução é válido para reuso, instruções de leitura e escrita verificam este sinalizador para reusarem os endereços de acesso à memória e instruções de leitura reusam o resultado da operação quando **mem valid** também estiver ativo. Instruções dependentes precisam ainda verificar se as últimas modificações realizadas nos registradores de entrada foram feitas pelo reuso das instruções fontes da dependência, e isto é feito consultando a *register source table*, **RST**, para verificar se as últimas instruções que atualizaram os registradores fontes são as mesmas instruções referenciadas pelos campos de índice dos operandos de entrada da instrução.

Invalidações ocorrem quando instruções de escrita à memória modificam o mesmo endereço de memória de uma instrução de leitura contida no *reuse buffer*, quando os registradores de entrada de instruções independentes são modificados, e quando a instrução fonte de uma cadeia de instruções dependentes é removida do *reuse buffer.*

Na **Figura 2.8** observamos um exemplo de funcionamento do mecanismo **Sn+d**, onde as instruções **100, 104** e **108** já foram executadas e inseridas no *reuse buffer.*

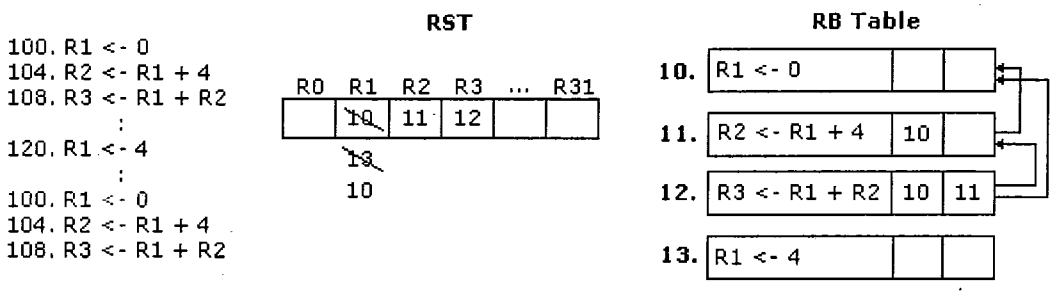


Figura 2.8: Estrutura básica de uma entrada do *result buffer*.

Quando a instrução **120** é executada o valor de **R1** é modificado e a entrada correspondente a ele na tabela **RST** recebe o valor **13** que representa a posição desta instrução no *reuse buffer*. Além disso, todas as entradas do *reuse buffer* que usam este registrador e não possuem dependência são invalidadas. Desta forma as entradas **11** e **12** da tabela de reuso, dependentes do resultado da entrada **10**, não serão invalidadas.

Se em seguida a instrução **100** é executada novamente o registrador **R1** será modificado e a entrada correspondente a ele na tabela **RST** receberá novamente o valor **10**. Deste modo quando a instrução **104** for acessada e a entrada correspondente no *reuse buffer* obtida, a análise da instrução **104** constatará que o valor do registrador fonte **R1** desta instrução é dependente da instrução localizada na posição **10** do *reuse buffer* e comparando este valor com o valor presente na tabela **RST**, que identifica a última instrução presente do *reuse buffer* que modificou o registrador **R1**, teremos que os valores são iguais e, portanto a instrução **104** também será reusada. De forma similar, a instrução **108** também será reusada, pois os seus operandos fontes dependem das instruções reusadas **100** e **104** [33].

2.3.2 COMPARANDO OS ESQUEMAS Sv, Sn, Sn+d

As **Figuras 2.9a**, **2.9b** e **2.9c** apresentam os percentuais de reuso de instruções obtidos pelos esquemas **Sv**, **Sn** e **Sn+d** usando um *reuse buffer* com 32, 128 e 1024 entradas, na execução de um *benchmark* composto por cinco programas do **SPECint '92** (**gcc**, **compress**, **eqntott**, **espresso** e **xlisp**), cinco programas do **SPECint '95** (**go**,

m88ksim, **vortex**, **ijpeg** e **perl**) e mais dois programas de processamento intensivo de operações com inteiros (**Yacr2** e **Mpeg**).

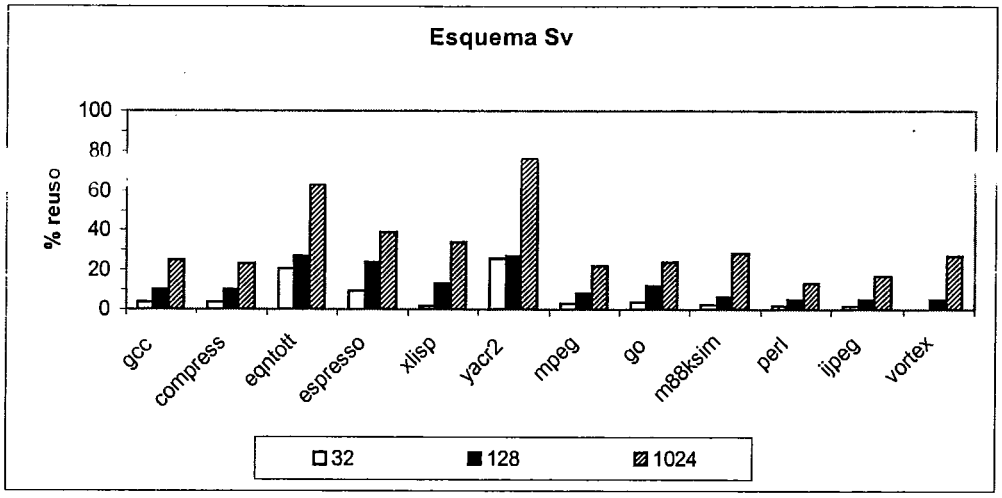


Figura 2.9a: Percentual de reuso de instruções com o esquema **Sv**.

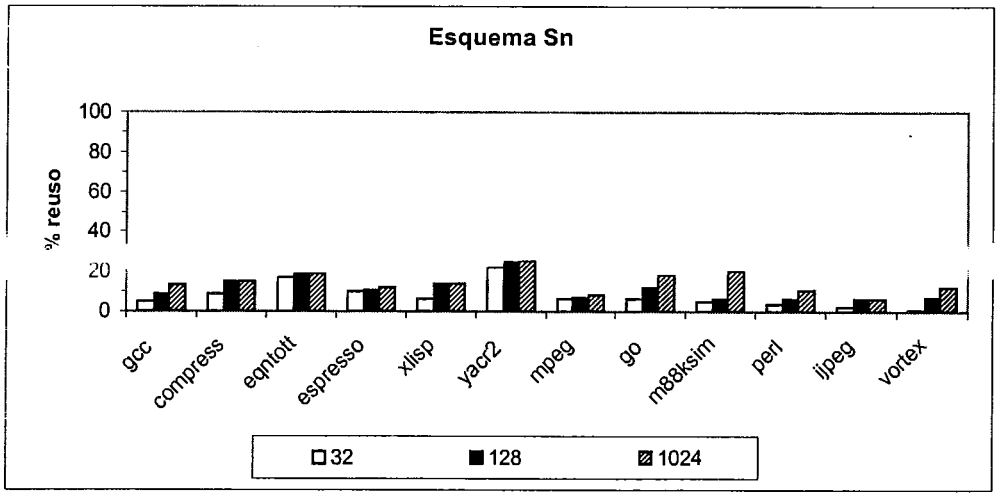


Figura 2.9b: Percentual de reuso de instruções com o esquema **Sn**.

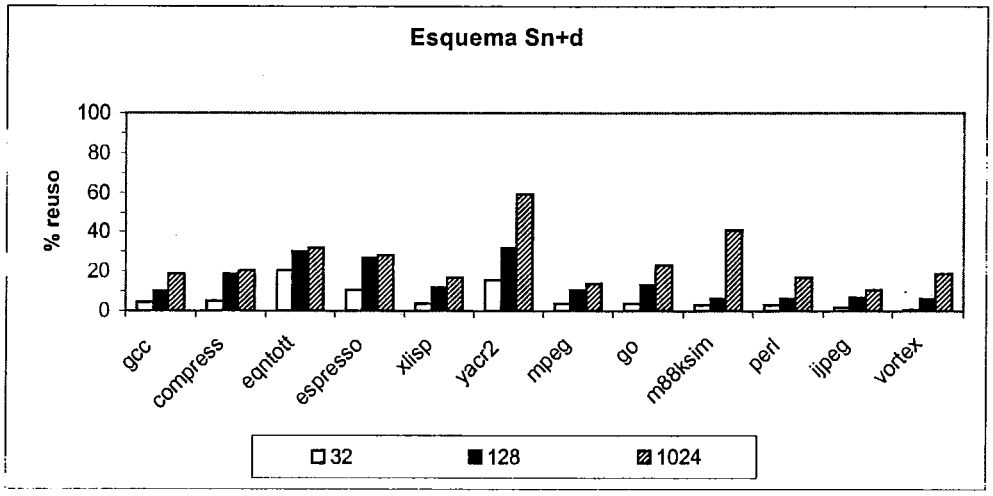


Figura 2.9c: Percentual de reuso de instruções com o esquema **Sn+d**.

Conforme podemos observar todos os mecanismos apresentaram um resultado significativo no reuso de instruções com taxas de até 76% para o esquema **Sv** e de até 59% para o esquema **Sn+d**. Porém, o esquema **Sv** não efetua verificações de dependências entre as instruções no momento do despacho e, portanto os dados reportados em [33] representam o potencial deste mecanismo e não uma implementação prática.

Deste modo, entre os mecanismos de implementação prática, **Sn** e **Sn+d**, o esquema **Sn+d** apresentou melhor percentual de reuso de instruções, devido à exploração do reuso de cadeias de dependências que reduziu a taxa de invalidações das instruções contidas no *reuse buffer* e aumentou a granularidade do mecanismo.

Outro importante resultado obtido em [33] diz respeito à contribuição que cada tipo de instrução traz ao mecanismo de reuso. A **Tabela 2.1** apresenta o percentual de reuso por tipo de instrução para um *reuse buffer* com 1024 entradas.

Categorias de Instruções	Sv	Sn	Sn+d
Leituras da Memória (valor)	21.2	6.7	11.9
Calculos de Endereçamento	20.8	10.3	11.9
Transferências de Controle	35.1	1.8	20.2
Instruções com dois operandos	36.9	26.8	32.0
Instruções com um operando	38.6	17.9	30.9
instruções com valor imediato	51.0	98.1	90.4

Tabela 2.1: Percentual de reuso por tipo de instrução.

Na **Tabela 2.1**, podemos observar que operações de leitura da memória representam 11.9% do total de instruções reusadas no esquema **Sn+d** e 21.2% no esquema **Sv**, mostrando que existe ainda um grande potencial de reuso à ser explorado. Além do reuso do valor de retorno das instruções de leitura da memória, o reuso do cálculo de endereçamento também representa um grande potencial de reuso a ser explorado.

Vantagens

Os mecanismos baseados em técnicas de previsão de valores introduzem custos adicionais para verificação e validação dos dados, além disso, as instruções ainda são executadas para verificação do resultado previsto utilizando os recursos do processador. Enquanto o uso de técnicas de reuso de valores possibilitam a antecipação dos resultados das instruções redundantes sem que haja necessidade de executá-las, minimizando a utilização de recursos do processador.

Outra vantagem destas técnicas é que não existem execuções especulativas com base nos valores retornados pelo reuso e, portanto não existem penalidades impostas a estes mecanismos.

Desvantagens

No esquema **Sn+d**. as instruções pertencentes a uma cadeia de dependências só serão reusadas no mesmo ciclo se todas as instruções estiverem contidas na janela de despacho, e, além disso, para manter a relação de dependência este esquema precisa conter todas as instruções pertencentes a uma cadeia de dependência dentro do *reuse buffer*. sendo que se uma instrução fonte da cadeia de dependência for invalidada todas as instruções seguintes serão invalidadas.

Devido à política de invalidação, os mecanismos **Sn** e **Sn+d** não possuem habilidades de manter várias instâncias de uma mesma instrução no *reuse buffer*.

Também nos esquemas apresentados, as instruções são inseridas no *reuse buffer* assim que são finalizadas, e isto torna o mecanismo bastante sensível à política de substituição implementada.

Nesta seção apresentamos um mecanismo que explora a localidade de valores pelo reuso dinâmico de instruções. Na próxima seção veremos um novo mecanismo que estende o conceito de reuso de valores ao nível de blocos básicos, aumentando a granularidade das técnicas de reuso.

2.4 REUSO DE BLOCOS BÁSICOS

Experimentos conduzidos em [16] demonstram que os valores de entrada e saída de blocos básicos apresentam considerável nível de localidade de valores. Além disso, estes experimentos constataram que para a totalidade dos programas analisados 90% dos blocos básicos mantiveram o número de registradores de entrada inferior a quatro registradores e o número de registradores de saída inferior a cinco, e que as quantidades de instruções de leitura e escrita na memória são respectivamente menores que quatro e cinco instruções deste tipo por bloco.

Estes resultados favorecem a exploração de técnicas de previsão e reuso de valores com mecanismos de maior granularidade e em [16] foi introduzido o mecanismo de *block reuse*, que estende o reuso de valores ao nível de blocos básicos.

2.4.1 IMPLEMENTAÇÃO

O mecanismo proposto em [16], consiste de uma tabela denominada *block history buffer*, ou **BHB**, que é responsável por armazenar as instâncias de execução dos blocos básicos.

Cada entrada da **BHB** consiste de um **tag** de identificação da entrada, formado pelo endereço da primeira instrução presente no bloco básico. Além dos contextos de entrada e saída do bloco, dos campos correspondentes às instruções de entrada e saída de memória, e de um ponteiro para o endereço da primeira instrução do próximo bloco básico, **Figura 2.10**.

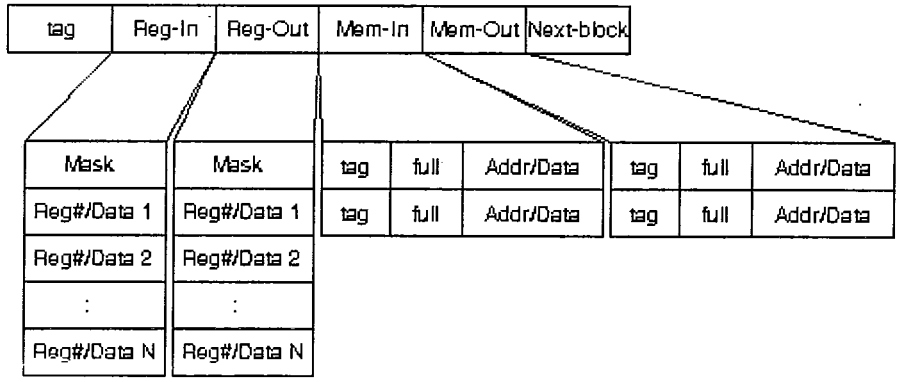


Figura 2.10: Estrutura básica de uma entrada do *block history buffer*.

Os campos que armazenam o contexto de entrada e saída, representados por **reg-in** e **reg-out** na figura, são formados pelos registradores e seus valores de dados, enquanto os campos **mem-in** e **mem-out** armazenam as operações de leitura e escrita à memória respectivamente e são compostos por um **tag** que identifica a instrução de memória através do próprio endereço da instrução, de um sinalizador, **full/empty**, que

indica se a entrada é valida, além do endereço de acesso à memória e do valor a ser lido ou armazenado.

Reuso de Blocos Básicos

Quando o endereço de inicio de um bloco básico é obtido, a **BHB** é consultada a procura das instâncias deste bloco que satisfaçam os valores de entrada. Assim que as instâncias são obtidas, os valores dos registradores de entrada são comparados com os valores armazenados, além disso se existem entradas válidas presentes em **mem-in** a memória *cache* é acessada e o valor armazenado também é comparado com o valor da *cache* para determinar o reuso do bloco. Se os valores comparados não corresponderem ou se o endereço de acesso à memória de uma das instruções em **mem-in** não está presente na *cache*, então o bloco não é reusado e as instruções são despachadas normalmente.

Quando uma instância de um bloco básico é reusada, os registradores de saída correspondentes são atualizados e as instruções pertencentes ao bloco são marcadas como completadas e são enviadas ao *reorder buffer* para finalização como qualquer outra instrução, porém as unidades funcionais não são utilizadas. Além disso, o endereço do próximo bloco básico, **next-block**, é usado para modificar o contador de programa, redirecionando-o para a primeira instrução do próximo bloco básico. Se o bloco contém uma instrução de escrita na memória, isto é, se existe alguma entrada válida em **mem-out**, o valor é armazenado na *cache* quando esta for finalizada.

Reconhecimento Dinâmico de Blocos Básicos

A identificação e construção dinâmica dos blocos básicos são realizadas da seguinte forma. Qualquer instrução que segue uma instrução de desvio é considerada começo de um novo bloco básico e uma instrução de desvio marca o fim do bloco básico. Se uma instrução de desvio redireciona o programa para uma instrução interna a um

bloco básico, uma nova entrada é criada na **BHB** começando no endereço da instrução alvo do desvio.

O reconhecimento dos registradores de entrada e saída de um bloco básico é realizado com o auxílio de duas mascaras de *bits* que identificam os registradores de entrada e de saída dinamicamente. Quando uma instrução é executada todos os *bits* da mascara de entrada correspondentes aos registradores fontes usados pela instrução e que não estão habilitados na mascara de *bits* de saída são ativados. Os *bits* da mascara de saída são habilitados para os registradores destino da instrução.

Para minimizar o número de registradores pertencentes ao contexto de saída de um bloco básico este mecanismo usa recursos de compilação para identificar *dead outputs*, isto é, valores produzidos pelo bloco básico e que não serão mais necessários.

2.4.2 COMPARANDO O MECANISMO

O mecanismo de *block reuse* aumenta a granularidade do mecanismo de reuso apresentado por [33], porém as seqüências de instruções são limitadas ao tamanho do bloco básico e a limitação no tamanho médio dos blocos básicos, encontrados em aplicações de inteiros, podem minimizar os ganhos com este mecanismo.

Porém blocos básicos com muitas instruções tendem a ter contextos de entrada grandes e que não podem ser mantidos inteiramente em uma única entrada. Além disso, blocos básicos grandes podem conter instruções pouco redundantes que reduzem a freqüência de reuso dele.

A técnica de reuso de blocos básicos adiciona instruções de acesso à memória diretamente nas entradas da **BHB**, desta forma o contexto de entrada, que precisa ser avaliado para reuso do bloco, se torna muito grande prejudicando os resultados obtidos. Além disso, a avaliação dos valores de memória é realizada comparando-os com os

valores correspondentes contidos na *cache*, o que induz o aumento no número de portas de leitura da memória *cache* para implementação deste mecanismo.

A **Tabela 2.2** a seguir apresenta os resultados obtidos em [16] com a implementação de diferentes configurações do mecanismo de reuso de blocos básicos usando o simulador **Simplescalar Tool Set** [1], executando um subconjunto de programas de *benchmarks* formado por três programas do **SPECint '92, alvinn, ear** e **wordcount**, e seis programas do **SPECint '95, compress, go, ijpeg, li, m88ksim** e **perl.**

	r-in=4,r-out=4 m-in=3,m-out=2	r-in=4, r-out=5 m-in=4,m-out=2	r-in=5,r-out=6 m-in=4,m-out=3	Ilimitado
alvinn	1.04	1.05	1.05	1.15
compress	1.06	1.06	1.06	1.07
ear	1.01	1.01	1.01	1.01
go	1.08	1.09	1.09	1.18
ijpeg	1.01	1.08	1.08	1.11
li	1.09	1.09	1.10	1.14
m88ksim	1.09	1.09	1.10	1.21
perl	1.12	1.14	1.16	1.37
wordcount	1.04	1.04	1.04	1.04

Tabela 2.2: *Speedups* para diferentes configurações de máquinas.

Em [17], a verificação de que a freqüência de reuso de blocos poderia ser mais bem explorada com a redução do contexto de entrada fez surgir um novo mecanismo denominado *sub-block reuse*, que procura reusar grupos de instruções com menor granularidade porém com maior freqüência de reuso.

Neste capítulo foi apresentado o conceito de localidade de valor que é o conceito base de todos os mecanismos de previsão e reuso de dados, além de alguns dos principais mecanismos que exploram tal conceito. No próximo capítulo iniciaremos a exploração da técnica de reuso de *traces* de instruções denominada **DTM** que utiliza dois níveis de tabelas de reuso para formar *traces* com melhor relação entre número de instruções e freqüência de reuso.

Capítulo 3

Memorização Dinâmica de *Traces* - DTM

Experimentos conduzidos em [3] introduziram o **DTM**, *Dynamic Trace Memoization*, uma técnica que estende o reuso de instruções para além das fronteiras dos blocos básicos. Este mecanismo utiliza duas tabelas de memorização na identificação dinâmica de seqüências de instruções redundantes.

O uso de duas tabelas de memorização fornece ao mecanismo dois níveis de reuso, o primeiro nível é usado no reuso de instruções simples e na construção de *traces* de instruções redundantes, enquanto o outro é responsável pelo reuso de *traces* de instruções. Os dois níveis de reuso também fornecem ao mecanismo uma volatilidade menor que a encontrada em outras técnicas semelhantes. Assim, quando uma instrução é executada pela primeira vez ela é inserida na tabela de memorização global, **MEMO_TABLE_G**. e somente as instâncias das instruções que estão presentes nesta tabela e que são reusadas irão constituir *traces* de instruções redundantes que são adicionados à tabela de memorização de *traces*, **MEMO_TABLE_T**.

Este tipo de implementação também fornece uma pré-qualificação das instruções que compõem um *trace*, porém não é um indicador de qualidade do *trace* formado, pois para qualificar um *trace* precisamos levar em consideração o número de instruções que estão contidas no *trace* e que pertencem ao caminho crítico de execução do programa, e a freqüência de reuso do *trace* durante a execução do programa.

Para auxiliar a compreensão deste mecanismo, na próxima seção apresentaremos as terminologias básicas introduzidas por esta técnica.

3.1 TERMINOLOGIA BÁSICA

A técnica de reuso de *traces* introduziu novos termos [3,8] que são usados na descrição do **DTM**, apresentado neste capítulo, e na descrição das variações deste mecanismo, que incluem instruções de acesso à memória e serão apresentadas nos capítulos seguintes.

***trace* de instruções** - Um *trace* de instruções, ou simplesmente *trace*, é uma seqüência dinâmica de instruções de um programa.

***trace* redundante** - Um *trace* é considerado redundante se todas as instruções pertencentes a ele forem redundantes.

instruções válidas - O **DTM** trabalha com um subconjunto das instruções do processador e as instruções que pertencem a este subconjunto são denominadas instruções válidas. Neste subconjunto não estão incluídas instruções de acesso à memória, chamadas ao sistema operacional e instruções de ponto flutuante. Embora instruções de acesso à memória não pertençam ao conjunto de instruções válidas o cálculo do endereço de acesso à memória é reusado pelo mecanismo.

contexto de entrada - Conjunto formado pelos registradores e respectivos valores de entrada das instruções presentes no *trace* e que são fornecidos por instruções externas a ele.

contexto de saída - Conjunto formado pelos registradores e respectivos valores de saída das instruções presentes no *trace*.

3.2 ESTRUTURA DAS TABELAS DE MEMORIZAÇÃO

A **Figura 3.1** apresenta o formato das entradas da **MEMO_TABLE_G**. As entradas desta tabela possuem como chave o endereço da instrução armazenado no campo **pc**, os campos **sv1** e **sv2** representam os valores dos operandos fontes das instruções e o campo **res/npc** compartilha o armazenamento do endereço alvo de instruções de desvio com o resultado de instruções lógicas e aritméticas.

Além destes campos são incluídos também três sinalizadores usados para atualização do preditor de desvios. O sinalizador **jmp** indica uma instrução de desvio que usa o valor contido no campo **res/npc** como endereço alvo do desvio, **brc** indica instrução de desvio condicional e **btaken** sinaliza se o desvio condicional é tomado ou não tomado, **Figura 3.1**.

pc	jmp	brc	btaken	sv1	sv2	res/npc
30b	1b	1b	1b	32b	32b	32b

pc – endereço da instruçao
jmp – sinalizador de instrução de desvio incondicional
brc – sinalizador de instrução de desvio condicional
btaken – sinalizador de desvio **tomado** ou **nao_tomado**
sv1 – valor do operando fonte 1
sv2 – valor do operando fonte 2
res/npc – resultado do oparacao/destino do desvio

Figura 3.1: Formato das entradas na **MEMO_TABLE_G**.

A **Figura 3.2** apresenta o formato das entradas da **MEMO_TABLE_T**. Esta tabela é acessada pelo endereço da primeira instrução do *trace* e o campo de identificação **pc** armazena o valor do índice de acesso.

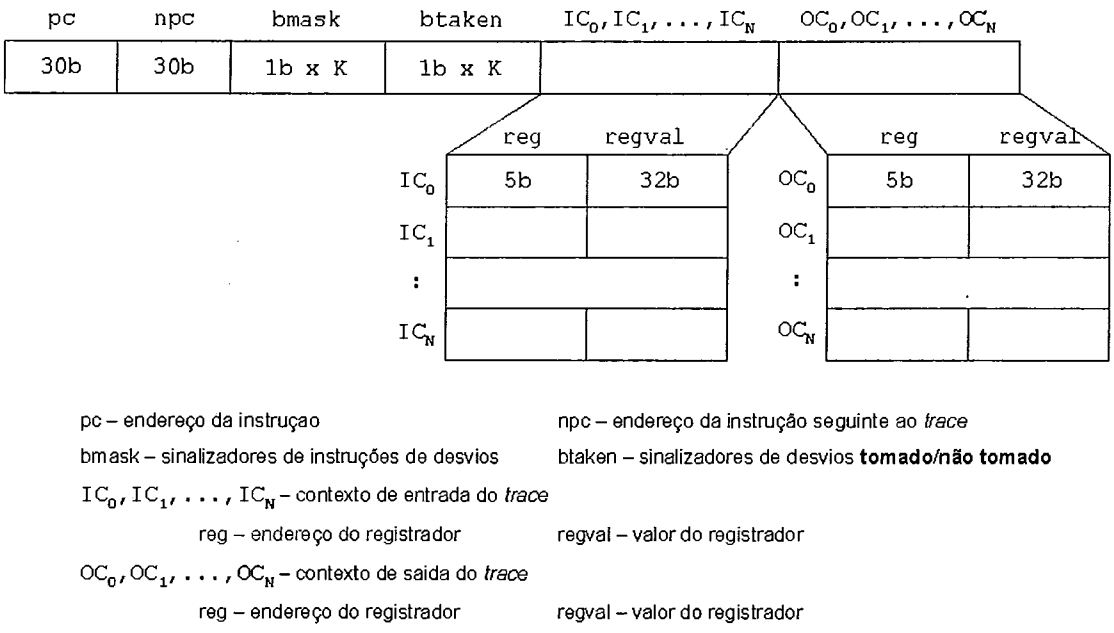


Figura 3.2: Formato das entradas na **MEMO_TABLE_T**.

Além do campo de identificação, cada entrada possui um campo que armazena o endereço da instrução seguinte ao *trace*, **npc**, um conjunto de campos que armazenam os contextos de entrada, $\mathbf{IC_N}$, e são constituídos pelos pares registrador/valor, por um conjunto de pares registrador/valor que armazenam os contextos de saída, $\mathbf{OC_M}$, e dois mapas de *bits*, **bmask** e **btaken**, que indicam a presença de instruções de desvios e informam para cada desvio se ele foi ou não tomado [3,8].

3.3 FUNCIONAMENTO DO MECANISMO

Quando uma instrução é acessada, é verificado se ela pertence ao conjunto de instruções válidas do **DTM**. Instruções que não são válidas são marcadas como **não_redundante** e seguem o caminho normal de execução.

Se uma instrução válida do **DTM** é acessada, as tabelas de memorização são consultadas para verificar se existe alguma instância de execução desta instrução ou se existe algum *trace* iniciando por ela. Se nenhuma instância de execução da instrução ou *trace* for encontrado nas tabelas de reuso, a instrução é marcada como **não_redundante**

e segue o caminho normal de execução. Entretanto, se alguma instância de execução da instrução for encontrada na **MEMO_TABLE_G** a instrução é marcada como **redundante** e as instâncias correspondentes são selecionadas.

No momento do despacho, se a instrução está marcada como **redundante**, os operandos fontes são acessados e seus valores comparados com os respectivos valores de entrada das instâncias selecionadas. Se estes valores forem iguais, a instrução é identificada como **redundante** e a instância correspondente reusada. Quando uma instrução é reusada os registradores destino são atualizados e a instrução é enviada diretamente ao *buffer* de reordenação. Porém, se os valores dos operandos de entrada não coincidirem com os valores dos operandos correspondentes nas instâncias selecionadas, a instrução é marcada como **não_redundante** e segue o caminho normal de execução. Instruções cujos operandos de entrada não estão disponíveis durante o teste de reuso, seguem o caminho normal de execução sendo marcadas como **não_redundante**.

Após a finalização da instrução, se ela pertence ao conjunto de instruções válidas do **DTM** e foi marcada como **não_redundante**, ela é inserida na **MEMO_TABLE_G**. Se a instruções foi marcada como **redundante**, ela é usada na construção de novos *traces*, e um *buffer* temporário com a mesma estrutura das entradas da **MEMO_TABLE_T** é usado para armazenar o *trace* em formação [3,8].

Quando uma instrução marcada como **não_redundante**, pela falta de operandos prontos na verificação do reuso, é finalizada. É realizada a comparação entre os valores dos operandos de entrada da instrução, obtidos após a execução, com os respectivos valores dos operandos das instâncias selecionadas, e se estes valores forem iguais à instrução é identificada como **redundante**, e incluída no *trace* em formação comforme apresentado em [8].

Para possibilitar que um *trace* seja finalizado e que no mesmo ciclo um novo *trace* comece a ser construído, o **DTM** usa dois *buffers* temporários [8], **Buffer_T#1** e

Buffer_T#2, que são chaveados quando um *trace* é finalizado e inserido na **MEMO_TABLE_T**, conforme apresentado na **Figura 3.3**.

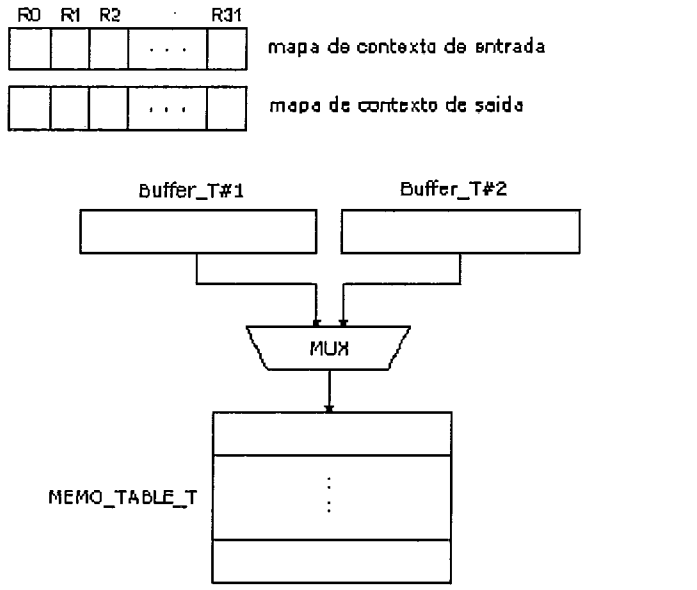


Figura 3.3: *Buffers* temporários e mapas de contextos de entrada e saída.

Quando uma instrução redundante é inserida no *buffer* temporário, o campo **npc** é modificado e passa a apontar para a instrução seguinte a ela no fluxo de execução. Além disso, os contextos de entrada e saída do *trace* e os sinalizadores de instruções de desvio são atualizados. Se a instrução inicia um novo *trace* o endereço da instrução é usado como índice para a **MEMO_TABLE_T** e o campo **pc** é modificado.

As identificações do contexto de entrada e saída são realizadas com o auxílio de dois mapas de *bits*, **Figura 3.3**. Quando uma instrução redundante vai ser adicionada a um *trace* em formação, o mapa de *bits* do contexto de saída é consultado para verificar se os valores de alguns dos operandos fontes da instrução são fornecidos por alguma instrução interna ao *trace*, e estando o *bit* correspondente a um destes registradores ativo, significa que o valor usado por este operando foi produzido por uma instrução interna ao *trace* e portanto, tal registrador não faz parte do contexto de entrada. Entretanto para os operandos fontes que não estão presentes no mapa de *bits* do contexto de saída é realizada a consulta ao mapa de *bits* do contexto de entrada e os registradores que não estiverem marcados são adicionados junto com os respectivos valores ao contexto de

entrada do *trace* e os sinalizadores correspondentes a eles são ativados no mapa de *bits* do contexto de entrada.

O contexto de saída de um trace é obtido verificando se os *bits* correspondentes aos registradores de saída da instrução estão habilitados. Se os sinalizadores não estiverem habilitados os registradores são adicionados ao contexto de saída do *trace* e os *bits* correspondentes no mapa de *bits* do contexto de saída são ativados. Caso os sinalizadores estejam habilitados as entradas correspondentes do contexto de saída do *trace* são atualizadas.

A construção de um *trace* termina quando uma instrução não redundante for encontrada, quando o contexto de entrada ou de saída do *trace* estiverem cheios ou quando uma instrução que inicia um *trace* reusado é encontrada. Se um *trace* em construção com mais de duas instruções for finalizado, ele é incluído na **MEMO_TABLE_T**, caso contrário o *trace* é descartado.

Quando uma instrução que inicia um *trace* é acessada, as tabelas de reuso são consultadas e se existirem instâncias de *traces* que iniciam no mesmo endereço da instrução, estes são selecionados para o teste de reuso. No estágio de despacho após a obtenção dos valores dos registradores fonte da instrução é realizada a verificação do contexto de entrada dos *traces* previamente selecionados e havendo um *trace* com contexto de entrada válido este é selecionado para reuso. Caso algum operando de entrada do *trace* não esteja pronto o *trace* não é reusado.

Quando um *trace* redundante é encontrado os registradores presentes no contexto de saída são atualizados e a instrução é adicionada ao *buffer* de reordenação, as instruções pertencentes ao *trace* e que estão presentes no *pipeline* são invalidadas, e o contador de programa é redirecionado para a instrução seguinte ao *trace*.

Exemplo de Construção de *Traces*

Suponha a seqüência de instruções redundantes apresentadas na **Figura 3.4**. Após a finalização da instrução de comparação no endereço **100**, e após a verificação de que esta instrução foi reusada, o mapa de *bits* do contexto de entrada é atualizado com a marcação dos registradores fontes da instrução que não estão marcados no mapa de *bits* do contexto de saída. Assim, o *bit* correspondente ao registrador **R1** é ativado, e ele é adicionado ao contexto de entrada do *buffer* temporário junto com o valor correspondente.

	MEMO_TABLE_G				contexto de entrada							contexto de saída						
	pc	sv1	sv2	res	r0	r1	r2	r3	r4	..	r31	r0	r1	r2	r3	r4	..	r31
100. cmp r2,r1,0x05	0x100	0x0006	-	0x0001		1								1				
104. bne r2,0x124	0x104	0x0001	-	0x124		1								1				
124. add r4,r3,0x02	0x124	0x0003	-	0x0005		1		1						1		1		
128. sll r2,r4,0x04	0x128	0x0005	-	0x0050		1		1						1		1		
12c. or r4,r4,r2	0x12c	0x0005	0x0050	0x0055		1		1						1		1		
130. sll r2,r4,0x08	0x130	0x0055	-	0x5500		1		1						1		1		
134. or r4,r2,r4	0x134	0x5500	0x0055	0x5555		1		1						1		1		

Buffer_T

pc	npc	bmask	btaken	IC_{r0}	IC_{v0}	IC_{r1}	IC_{v1}	IC_{r2}	IC_{v2}	OC_{r0}	OC_{v0}	OC_{r1}	OC_{v1}	OC_{r2}	OC_{v2}
0x100	0x104			r1	0x0006					r2	0x0001				
0x100	0x124	1	1	r1	0x0006					r2	0x0001				
0x100	0x128	1	1	r1	0x0006	r3	0x0003			r2	0x0001	r4	0x0005		
0x100	0x12c	1	1	r1	0x0006	r3	0x0003			r2	0x0050	r4	0x0005		
0x100	0x130	1	1	r1	0x0006	r3	0x0003			r2	0x0050	r4	0x0055		
0x100	0x134	1	1	r1	0x0006	r3	0x0003			r2	0x5500	r4	0x0055		
0x100	0x138	1	1	r1	0x0006	r3	0x0003			r2	0x5500	r4	0x5555		

Figura 3.4: Exemplo de construção dinâmica de *traces* com o **DTM**.

Em seguida, o mapa de *bits* do contexto de saída é consultado e se o *bit* correspondente ao registrador destino da instrução, registrador **R2**, não estiver habilitado, o registrador e seu valor são adicionados a uma entrada no contexto de saída do *buffer* temporário, e a entrada correspondente no mapa de *bits* do contexto de saída é ativada. Para finalizar, o campo **npc** é atualizado com o valor do endereço da instrução **104**, seguinte ao *trace*.

Após o término de execução da instrução **104** inicia-se o processo para adição desta instrução ao *trace*. Para isso, o mapa de *bits* do contexto de saída é consultado para verificar se o registrador **R2** fonte da instrução é usado como registrador destino de alguma instrução anterior incluída no *trace*, como **R2** está presente no contexto de saída ele não é adicionado ao contexto de entrada do *trace* nem o sinalizador correspondente no mapa de *bits* do contexto de entrada é ativado.

A instrução **104** é uma instrução de desvio e o valor armazenado no campo **pc** do *buffer* temporário é atualizado com o endereço alvo do desvio e as mascaras de *bits* sinalizadores de desvios correspondentes são ativadas.

Em seguida a instrução **124** é concluída e o mapa de *bits* do contexto de saída é consultado para verificar se o registrador fonte, **R3**, é usado como destino por alguma instrução anterior pertencente ao *trace*. Como o sinalizador correspondente não está habilitado, este registrador é adicionado ao contexto de entrada e o campo correspondente no mapa de *bits* do contexto de entrada é habilitado. O registrador **R4** destino desta instrução também não se encontra no mapa de *bits* do contexto de saída do *trace* e desta forma é adicionado a uma entrada do contexto de saída do *trace* em construção e a entrada correspondente no mapa de *bits* é ativada.

O processo de construção de *trace* prossegue até que uma instrução não redundante é encontrada. Quando o *trace* finalizado possui mais de uma instrução ele é adicionado a **MEMO_TABLE_T**, caso contrário ele é descartado.

3.4 IMPLEMENTAÇÃO DO DTM

A implementação do mecanismo **DTM** é dividida em três estágios, o estágio **DS1** que identifica instruções válidas e acessa as tabelas de reuso a procura de instâncias de execuções anteriores da instrução e por *traces* iniciados por ela. O estágio **DS2**, onde os operandos de entrada das instruções e dos *traces* selecionados pelo estágio **DS1** são comparados com os valores dos registradores. Havendo igualdade entre os operandos, o

trace é reusado. Caso contrário, os valores dos operandos fontes das instâncias da instrução que foram selecionadas são verificados e havendo igualdade a instrução correspondente é reusada. A construção dos *traces* é realizada no estágio **DS3** que retira instruções reusadas do topo do *buffer* de reordenação e adiciona ao *trace* em formação [8]. A **Figura 3.5** apresenta o diagrama de blocos de uma arquitetura superescalar com **DTM**.

O **DTM** efetua duas tarefas básicas: Construção de *traces* através do reconhecimento de instruções redundantes, e reuso de traces. Trataremos nas próximas seções da implementação destas tarefas separadamente.

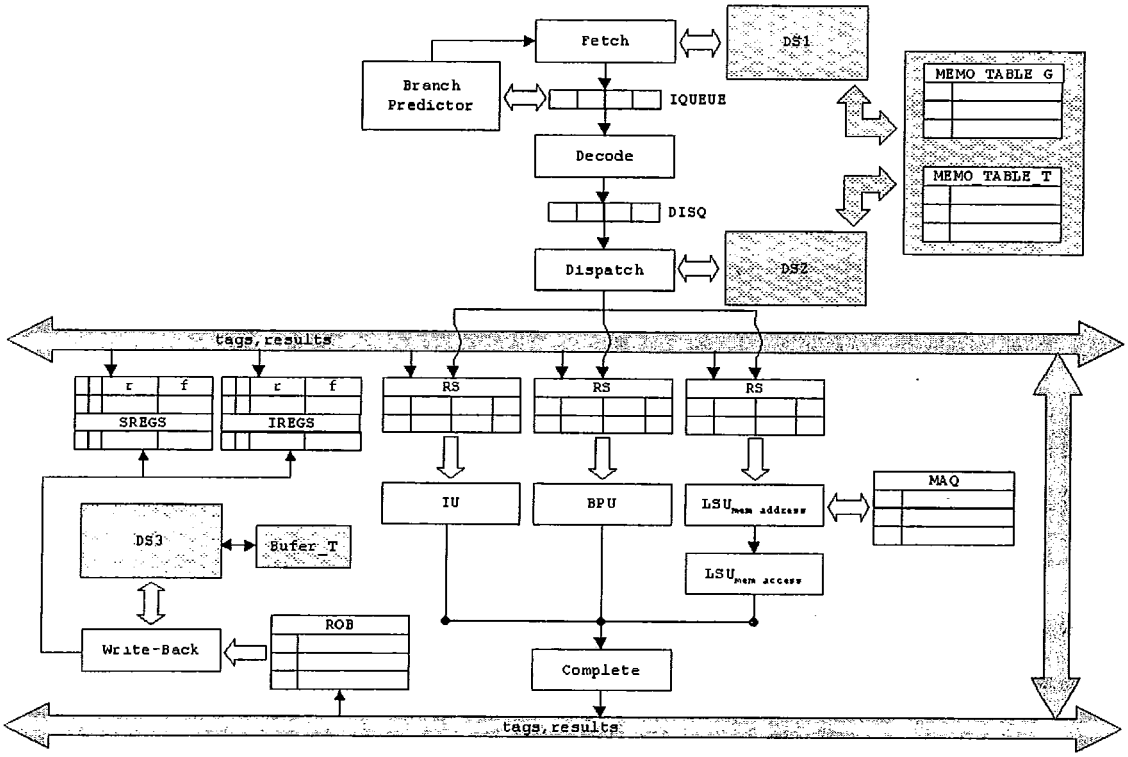


Figura 3.5: Diagrama de blocos de um processador superescalar com **DTM**.

3.4.1 CONSTRUÇÃO DE *TRACES*

Durante o estágio de busca as instruções são acessadas e simultaneamente a **MEMO_TABLE_G** é consultada a procura de instâncias de execuções anteriores das instruções. Se existem instâncias de execuções anteriores, estas são selecionadas para

verificação de redundância no estágio de despacho. Se a instrução é redundante, ela é enviada ao *buffer* de reordenação, **ROB**, e os registradores futuros são atualizados com os valores de saída correspondentes. Caso seja uma instrução de leitura ou escrita em memória, ela é enviada a estação de reserva apropriada e a fila de acesso à memória (**MAQ**) antecipando o cálculo do endereçamento de memória.

A adição de novas instruções a **MEMO_TABLE_G** e a construção de *traces* ocorre no estágio de finalização quando as instruções são removidas do **ROB**. Isto evita a inclusão de instruções executadas especulativamente nas tabelas de reuso.

Se a instrução removida do **ROB** está marcada como **redundante**, os mapas de *bits* do contexto de entrada e saída são consultados e o contexto de entrada e saída do *trace* em construção atualizado. Quando uma instrução válida e **não_redundante** é obtida do **ROB** a instância de execução da instrução é adicionada a **MEMO_TABLE_G** e o *buffer* em construção finalizado.

Quando uma instrução de leitura ou de escrita com endereço de acesso à memória reusado é encontrada, o *trace* em construção é finalizado e o valor do endereço e o tipo da operação de acesso à memória é armazenado no *trace*. Um *trace* em construção também será concluído se a instância da instrução obtida do *buffer* de reordenação representa um *trace* reusado. A **Tabela 3.1** descreve as ações tomadas nas etapas do processo de construção de *traces* [8].

ESTÁGIOS DO DTM	ESTÁGIOS	AÇÕES EXECUTADAS
DS1	*fetch*	- Seleciona as instâncias das instruções acessadas pelo mecanismo de busca que estão presentes na tabela de memorização global - MEMO_TABLE_G;
DS2	*dispatch*	- Verifica entre as instâncias selecionadas se existe alguma que satisfaz o teste de reuso; - Atualiza os valores dos registradores de saída usados pelas instruções reusadas;
DS3	*write-back*	- Insere instruções válidas e não redundantes na tabela de memorização global - MEMO_TABLE_G; - Instruções redundantes atualizam o mapa de *bits* do contexto de entrada e saída e são adicionadas ao *trace* em formação;

Tabela 3.1: Ações tomadas no processo de construção de *traces*.

3.4.2 REUSO DE *TRACES*

O processo de reuso de *traces* executa os seguintes passos: Inicialmente as instruções são acessadas no ciclo de busca e, simultaneamente a **MEMO_TABLE_T** é pesquisada a procura de *traces* de instruções que se iniciam no mesmo endereço. Se forem encontrados *traces* iniciando nos mesmos endereços, estes são selecionados para o teste de reuso.

No estágio de despacho as instâncias selecionadas são avaliadas e havendo alguma que atenda ao teste de reuso o *trace* é reusado. Quando um *trace* é reusado o seu contexto de saída é inserido no **ROB** e os registradores futuros são atualizados. Se o *trace* contiver uma operação de leitura ou escrita na memória finalizando-o, esta operação é inserida na **MAQ** e na estação de reserva da unidade de acesso à memória.

Após o término do reuso de um *trace*, ele é removido do **ROB** e os registradores reais que fazem parte do contexto de saída do *trace* são atualizados. A **Tabela 3.2**, à seguir, descreve as ações tomadas em cada etapa do processo de reuso de *traces* [8].

ESTÁGIOS DO DTM	ESTÁGIOS	AÇÕES EXECUTADAS
DS1	*fetch*	- Seleciona as instâncias dos *traces* que iniciam no mesmo endereço das instruções acessadas pelo mecanismo de busca e que estão presentes na tabela de memorização de *traces* - MEMO_TABLE_T;
DS2	*dispatch*	- Identifica traces redundantes comparando os contextos de entrada dos *traces* selecionados com os valores dos registradores; - Os valores do contexto de saída do *trace* redundante é usado para atualizar os registradores;
DS3	*write-back*	- Após a remoção do *trace* do *Reorder Buffer* os registradores da arquitetura são atualizados com os valores do contexto de saída do *trace;*

Tabela 3.2: Ações tomadas em cada etapa do processo de reuso de *traces*.

3.4.3 DETALHES DE IMPLEMENTAÇÃO

A implementação do **DTM** em uma arquitetura que usa o conjunto de instruções do **Sparc v7** induziu uma série de situações específicas deste tipo de máquina que não foram encontradas nos experimentos realizados em [8], e esta seção apresenta os detalhes desta implementação.

Delay Slot

A arquitetura **Sparc v7** usa um mecanismo de *delay slot* para minimizar as penalidades impostas por instruções de desvios, porém este tipo de implementação provoca alguns efeitos colaterais no **DTM**. Suponha a seqüência de instruções apresentadas na **Figura 3.6**, a seguir.

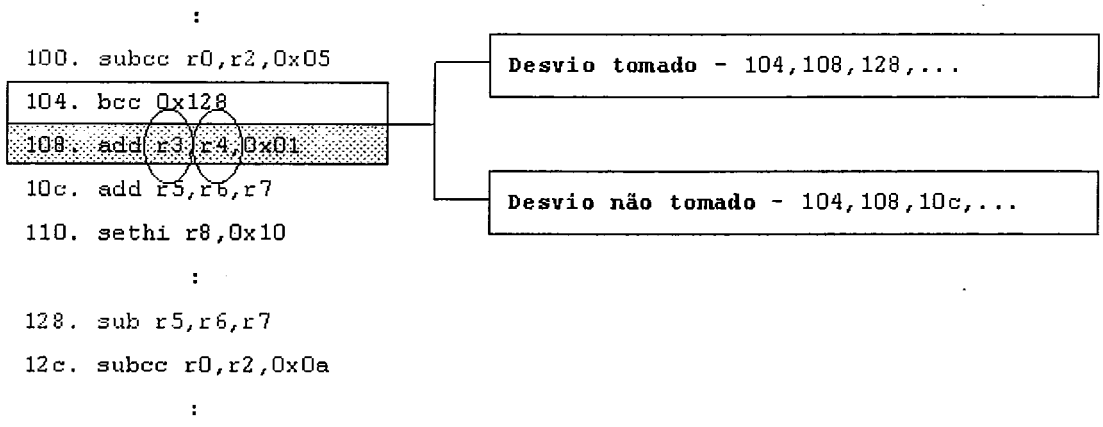


Figura 3.6: Trecho de código de programa para o **Sparc v7**.

Na **Figura 3.6**, a instrução **108** está ocupando o *delay slot* da instrução de desvio **104**, esta instrução será sempre executada independente do caminho seguido pelo programa. Deste modo podemos ter dois *traces* formados pelas instruções **108-10C** e **108-128** com os mesmos endereços de início, **108**, os mesmos contextos de entrada, **R4**, **R6** e **R7**, porém com diferentes endereços de destino gerando uma ambigüidade no mecanismo. A solução encontrada para contornar este problema foi eliminar a construção de *traces* que iniciam com instruções no *delay slot*.

De forma similar, o reuso de instruções simples que ocupam posições de *delay slot* produz ambigüidade no fluxo seguido pelo programa. Deste modo, estas instruções não são incluídas na **MEMO_TABLE_G**, o que impede o reuso delas e conseqüentemente induz a finalização de qualquer *trace* em construção.

Outro problema com instruções que ocupam um *delay slot* é que tais instruções acompanham sempre instruções de desvios e desta forma se um *trace* possui uma instrução de desvio, ele terá que carregar também a instrução que ocupa a posição de *delay slot* correspondente, mesmo que ela não seja redundante. Assim toda instrução que ocupa uma posição de *delay slot* finaliza o *trace* em formação, sendo incluída nele.

Janela de Registradores

O **Sparc v7** usa o conceito de janela de registradores para otimizar a passagem de parâmetros entre sub-rotinas de um programa. Está técnica implementa o deslocamento da janela de registradores sempre que instruções de *save* ou *retore* são executadas.

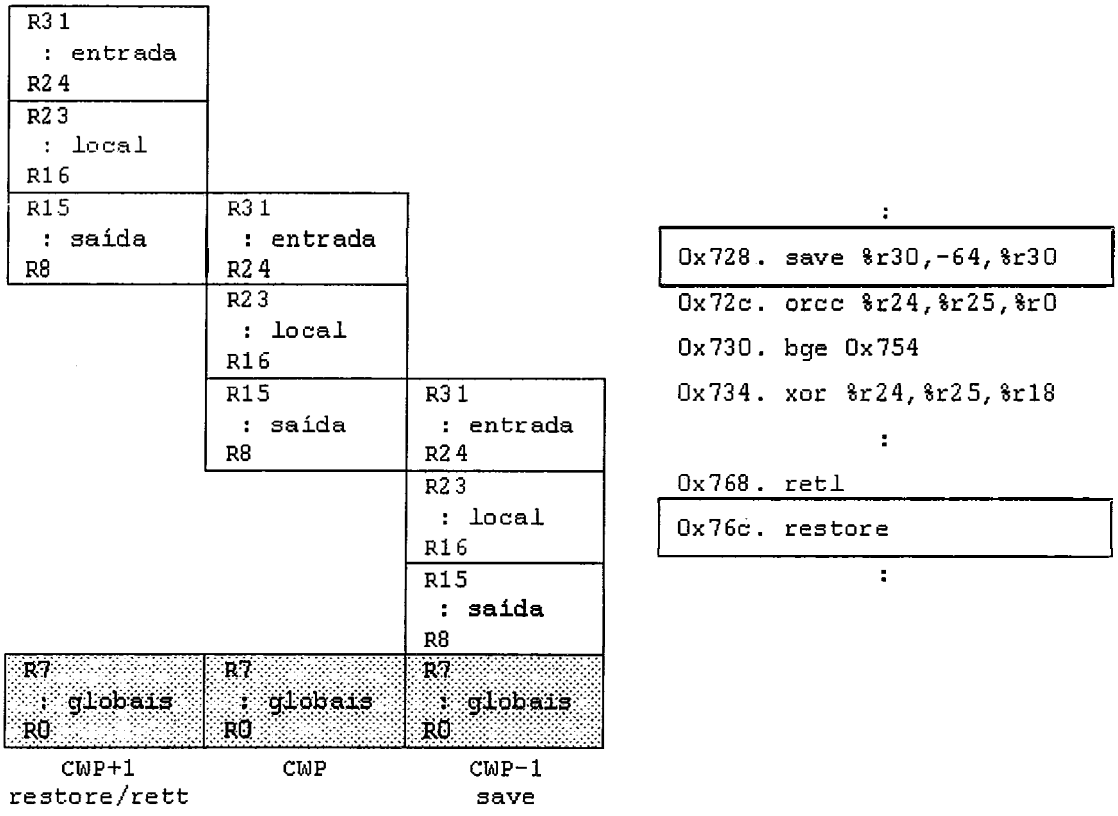


Figura 3.7: Ilustração da janela de registradores.

Conforme apresentado na **Figura 3.7**, um registrador é identificado pelo ponteiro para a janela corrente, **CWP**, e pelo endereço dele na janela. Para possibilitar o uso deste mecanismo em uma máquina que permite execução fora de ordem, as janelas de registradores passaram a ser movimentadas no estágio de despacho assim que uma instrução *save* ou *restore* é alcançada. Além disso, toda instrução carrega a janela fonte e destino de suas operações.

Existem duas abordagens distintas para a implementação desta técnica em uma máquina com execução fora de ordem. Podemos atribuir um identificador único para cada registrador da máquina e usar o valor do ponteiro **CWP** associado ao endereço do registrador para obter a posição dele no banco de registradores, deste modo cada instrução armazena os identificadores físicos de cada um dos operandos de entrada e saída. Outra forma de contornar este problema é atribuir o valor do ponteiro da janela de registradores dos operandos de entrada e dos operandos de saída em cada instrução. Assim sempre que um registrador da máquina for acessado será necessário enviar o ponteiro **CWP** junto com o endereço relativo do registrador dentro da janela.

A primeira vista as duas abordagens parecem similares, mas a implementação do **DTM** introduz diferenças significativas. O uso da solução que emprega um identificador único para cada registrador permite a inclusão de instruções de *save* e *restore* no conjunto de instruções válidas do mecanismo, porém o contexto de entrada dos *traces* estarão sempre vinculados ao endereço físico do registrador e não ao seu endereço na janela corrente e com isso *traces* que são redundantes dentro de rotinas recursivas não serão reusados, pois os identificadores físicos dos registradores serão diferentes embora os endereços relativos dentro da janela sejam os mesmos.

Assim a solução implementada foi utilizar o ponteiro **CWP** e o endereço do operando para identificar o registrador requerido. Desta forma cada *trace* armazena um ponteiro para a janela de registradores corrente e as instruções de *save* e *restore* são removidas do conjunto de instruções válidas.

Nesta seção apresentamos a implementação do **DTM** em uma arquitetura superescalar executando o conjunto de instruções do **Sparc v7**. Na próxima seção avaliaremos os resultados obtidos por [8] e faremos pequenas comparações com outros mecanismos de reuso de instruções.

3.5 COMPARANDO O DTM COM OUTRAS TÉCNICAS DE REUSO

O **DTM** oferece vários recursos que estendem as técnicas de reuso apresentadas anteriormente. Os resultados obtidos em [8] demonstram que as técnicas de reuso de instruções simples não exploram todo o potencial de reuso das aplicações. Isto pode ser observado no gráfico da **Figura 3.8** que apresenta o ganho de desempenho obtido com o crescente aumento do número de entradas da **MEMO_TABLE_T** para um mesmo tamanho da **MEMO_TABLE_G**.

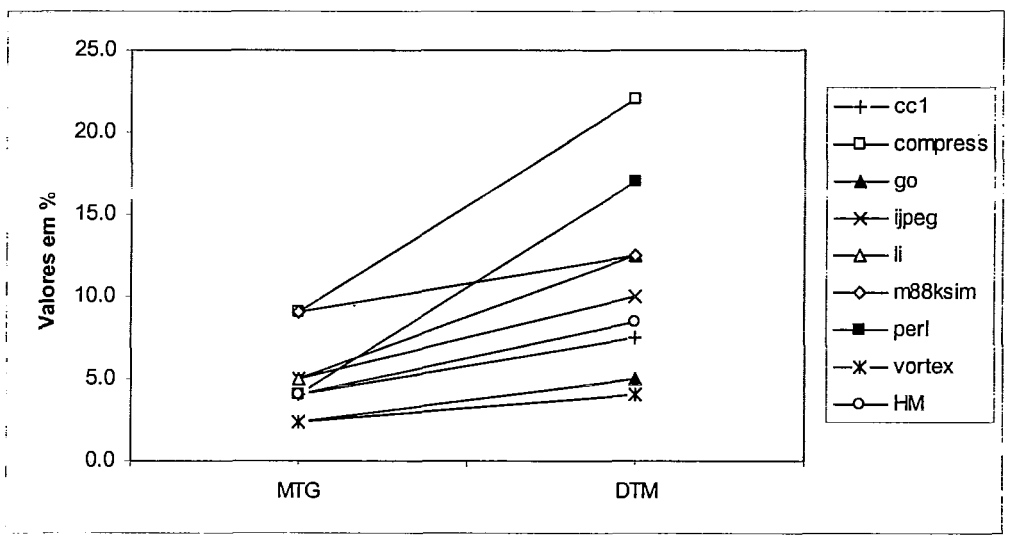


Figura 3.8: Aumento de desempenho com o crescimento da **MEMO_TABLE_T**.

A técnica de reuso de instruções, **Sn+d** [33], procura aproveitar este potencial de reuso existente com a exploração de cadeias de instruções dependentes, porém este mecanismo apresenta desvantagens significativas quando comparado com o **DTM**.

Para o mecanismo **Sn+d** somente instruções pertencentes a uma cadeia de dependência e que se encontram na fila de despacho podem ser reusadas em um mesmo ciclo. Além disso, este esquema é limitado pela largura de despacho de modo que quando instrução pertencente a uma cadeia de dependência é encontrada em um ponto **p** intermediário na fila de despacho que possui largura **L**, apenas **L - p** instruções poderão ser reusadas em um mesmo ciclo. No mecanismo **DTM** o número de instruções reusadas

independe da largura de despacho e desta forma mesmo que um *trace* esteja na última entrada da fila de despacho todas as instruções capturadas por ele serão reusadas.

No processo de reuso de cadeias de instruções o esquema **Sn+d** precisa armazenar todas as instruções pertencentes à cadeia de dependência, enquanto o **DTM** armazena apenas os contextos de entrada e saída dos *traces*. Desta forma, o aumento da cadeia de dependência tende a ocupar menos espaço nas tabelas de reuso. Além disso, o **DTM** implementa dois níveis de reuso o que mantém uma volatilidade menor na **MEMO_TABLE_T** com *traces* formados por instruções com melhor freqüência de reuso.

O **Sn+d** não reusa mais de uma instrução de desvio, se o desvio for previsto como tomado, pois a instrução alvo do desvio pode na maioria das vezes estar fora da janela de despacho, enquanto o **DTM** não possui limitações quanto ao número de desvios reusados por *trace*.

A **Figura 3.9** compara o reuso explorado pelo **DTM** com os resultados alcançados com o uso do esquema **Sn+d**, onde ambos os mecanismos foram configurados com a mesma capacidade de armazenamento [8]. Esta figura mostra que o reuso explorado pelo mecanismo **DTM** é maior que o obtido pelo mecanismo **Sn+d**.

Comparando o **DTM** com o mecanismo de reuso de blocos básicos apresentado no capítulo anterior, observamos diferenças significativas. Primeiramente os *traces* construídos pelo **DTM** são formados por instruções redundantes, isto é, por instruções que já foram executadas e reusadas pelo menos uma vez. Enquanto o mecanismo de reuso de blocos básicos apresentado por [16], adiciona todas as instruções contidas no bloco, o que reduz a tendência de reuso deste esquema. Em [17] os blocos básicos são divididos em partes menores que são mais bem explorados pelo mecanismo, porém também restrito às fronteiras dos blocos básicos.

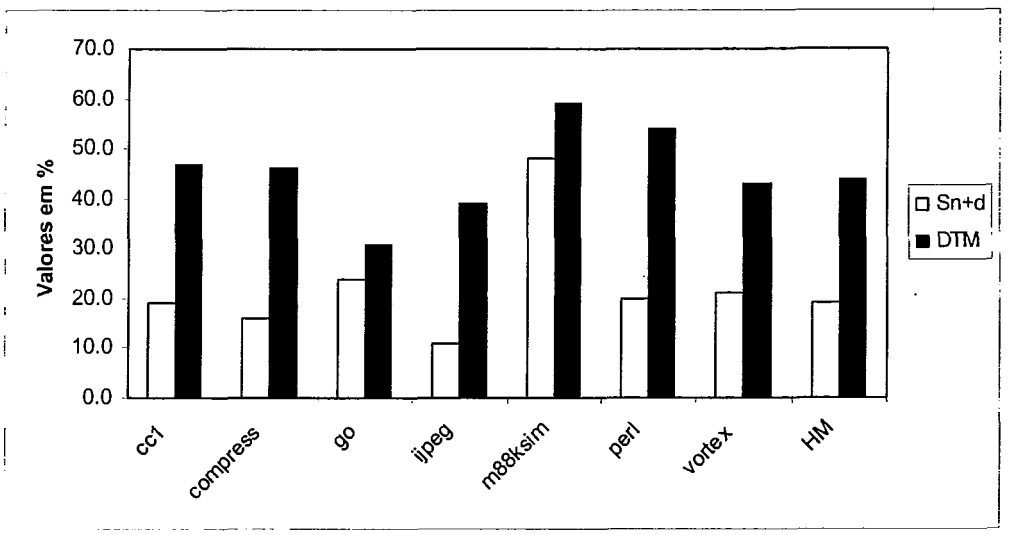


Figura 3.9: Comparação entre o reuso explorado pelo **DTM** e **Sn+d.**

O **DTM** não limita os tamanhos dos *traces* que podem atravessar o comprimento de vários blocos básicos. *Traces* que atravessam vários blocos básicos capturam o fluxo de execução correto da aplicação e isto favorece a taxa de acerto das previsões.

Neste capítulo apresentamos o **DTM** que é o mecanismo base deste trabalho e nos próximos capítulos mostraremos a implementação de reuso de instruções de memória neste mecanismo. No **Capítulo 6** mostraremos os resultados comparativos entre a implementação do **DTM** realizada em [8] e a efetuada neste trabalho, além da comparação entre as diferentes variantes deste mecanismo com reuso de valores de memória.

Capítulo 4

Adicionando Instruções de Acesso à Memória ao DTM

Experimentos realizados com o **DTM** em [3,8] mostram que o tamanho médio dos *traces* redundantes é de aproximadamente três instruções e que a dificuldade do mecanismo em obter traces de comprimentos maiores é atribuída à presença de instruções não reusáveis que surgem durante a construção dos *traces* e que implicam em sua finalização.

A quantidade de instruções não reusáveis encontradas em um programa depende diretamente das características dele. Instruções não válidas ao mecanismo provocam limitações no tamanho dos *traces* e podem contribuir para a indisponibilidade de operandos prontos. Deste modo, tratando um número maior de tipos de instruções torna-se possível o reuso de um número maior de instruções favorecendo o **DTM** e permitindo a antecipação de um número maior de valores de dados. Experimentos realizados com o **DTM** em uma arquitetura superescalar para o conjunto de instruções do **Sparc v7** executando os programas do **SPECint '95** mostram que em média 13% dos *traces* reusados são finalizados por instruções de acesso à memória e 36% deles por instruções não redundantes. Os experimentos também demonstram que muitos *traces* não são reusados por falta de operandos prontos no momento do teste no despacho, e isto é motivado pelas dependências de dados existentes entre os registradores requeridos no contexto de entrada dos *traces* e instruções despachadas que ainda não foram executadas. Estes experimentos foram realizados inicialmente excluindo do domínio de instruções

válidas às instruções de ponto flutuante, instruções de acesso à memória e chamadas ao sistema operacional.

Instruções de ponto flutuante são representativas em aplicações que fazem uso intensivo deste tipo de processamento, porém são praticamente desprezíveis em outros tipos de aplicações. As chamadas ao sistema operacional também são pouco freqüentes dentro de uma aplicação, sendo sua freqüência inferior a 0,6% [15]. Assim tanto as instruções de ponto flutuante quanto chamadas ao sistema operacional podem ser desprezadas em uma primeira abordagem.

Estudos anteriores mostram que as operações de acesso à memória representam 36% do total das operações executadas em um programa [15], e estas instruções foram responsáveis pela finalização de 13% dos *traces* identificados. Instruções de acesso à memória também são freqüentemente utilizadas no inicio de uma computação onde os valores dos dados são obtidos e usados na inicialização do processamento e no fim da computação quando os resultados do processamento precisam ser armazenados e enviados a outras áreas do programa. Assim instruções de acesso à memória estão freqüentemente servindo outras instruções e a inclusão delas no conjunto de instruções válidas do **DTM** pode contribuir para aumentar o tamanho médio dos *traces* e para minimizar o problema da indisponibilidade dos operandos de entrada.

4.1 CONDIÇÕES INICIAIS

Antes de prosseguirmos, precisamos estabelecer algumas condições iniciais para a inclusão de instruções de acesso à memória no **DTM**.

Inicialmente devemos adotar medidas para assegurar que operações de escrita e leitura em endereços de entrada e saída não sejam reusadas. Máquinas que usam instruções dedicadas às operações de entrada e saída não precisam de cuidados adicionais com o acesso destas operações, bastando que elas fiquem fora do conjunto de instruções válidas do mecanismo. Porém máquinas que fazem mapeamento de entrada e saída em

posições de memória precisam de um meio para delimitar esses endereços de memória e para isso definiremos que as arquiteturas que implementam os mecanismos apresentados nas seções seguintes, usam um par de registradores que delimitam a região de memória usada para mapear os endereços de entrada e saída ou que possuem instruções dedicadas a esta finalidade.

Operações de leitura em áreas de *buffers* de entrada e saída, também trazem problemas de inconsistência entre os valores destas regiões com os valores contidos nas tabelas de reuso e desta forma as regiões de memória usadas como áreas de *buffers* de dispositivos também precisam ser delimitadas. Devemos considerar também que as arquiteturas implementadas nas seções seguintes, possuem instruções que habilitam e desabilitam os mecanismos, assim como instruções que efetuam *flush* nas tabelas de reuso. Tais instruções podem ser usadas pelo sistema operacional com a finalidade de evitar o uso do mecanismo de reuso de instruções em áreas de processamento crítico.

Agora que definimos algumas condições iniciais para a implementação do mecanismo, veremos na próxima seção algumas propostas preliminares.

4.2 MODIFICAÇÕES IMPLEMENTADAS NO DTM

A implementação de instruções de acesso à memória no **DTM** levou a duas modificações significativas no mecanismo original. A primeira modificação foi realizada nas tabelas de reuso a fim de fornecer suporte para instruções de leitura e escrita na memória. A outra modificação foi efetuada no caminho do *pipeline* seguido pelas instruções e *traces* redundantes que possuem instruções de acesso à memória.

A **Figura 4.1**, apresenta as modificações realizadas nas entradas da **MEMO_TABLE_G**, onde o campo **maddr** armazena o endereço de acesso à memória de instruções de *load* e *store* e o campo **res/npc** assume uma nova função além das atividades de armazenar resultado e endereço alvo de instruções de desvio, este campo passa a armazenar também o valor a ser lido e escrito em memória por instruções de *load*

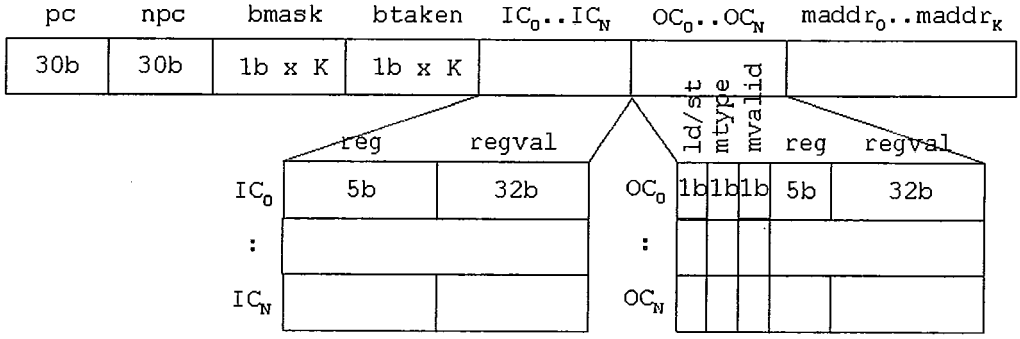
e *store* reusadas. Além disso, um sinalizador **mem valid** é usado para indicar se o valor armazenado no campo **res/npc** é válido para reuso por uma instrução de leitura da memória.

pc	jmp	brc	btaken	sv1	sv2	maddr	mem valid	res/npc
30b	1b	1b	1b	32b	32b	32b	1b	32b

pc – endereço da instruçao
jmp – sinalizador de instrução de desvio incondicional
brc – sinalizador de instrução de desvio condicional
btaken – sinalizador de desvio **tomado** ou **nao_tomado**
sv1 – valor do operando fonte
sv2 – valor do operando fonte
maddr – armazena o endereço de acesso à memória
mem valid – sinalizador de valor de leitura de memória válido
res/npc – resultado da oparacao/endereço alvo de desvio/valor à ser lido e armazenado em memória

Figura 4.1: Entradas da **MEMO_TABLE_G** com suporte a *load/store*.

Na **Figura 4.2** temos as modificações realizadas na **MEMO_TABLE_T**, onde dois novos conjuntos de campos foram adicionados às entradas desta tabela. O primeiro conjunto armazena o tipo das instruções de acesso à memória que serão reusadas, enquanto o segundo conjunto armazena os endereços de acesso à memória conforme pode ser visto na **Figura 4.2** a seguir.



pc – endereço da instruçao
npc – endereço da instrução seguinte ao *trace*
bmask – sinalizadores de instruções de desvios
btaken – sinalizadores de desvios **tomado/não tomado**
$IC_0, \ldots, IC_N$ – contexto de entrada do *trace*
reg – endereço do registrador
regval – valor do registrador
$OC_0, \ldots, OC_N$ – contexto de saida do *trace*
reg – endereço do registrador
regval – valor do registrador
ld/st - sinalizador de instrução de acesso à memória
mtype - sinalizador de tipo de acesso à memória (load/store)
mvalid - sinalizador de valor de memória válido para reuso
$maddr_0, \ldots, maddr_K$ - endereço de acesso à memória

Figura 4.2: Entradas da **MEMO_TABLE_T** com suporte a *load/store*.

Além dos conjuntos de campos apresentados na figura, foram adicionados também alguns sinalizadores ao contexto de saída do *trace*, que possibilitaram o uso destes campos no armazenamento do valor a ser lido ou escrito em memória.

Para adicionar instruções de acesso à memória ao **DTM** também foram realizadas algumas modificações no percurso seguido pelas instruções dentro do *pipeline*. Quando uma instrução ou *trace* reusado contendo instruções de acesso à memória é encontrado dois caminhos distintos podem ser tomados. Se a instrução ou *trace* contém operações de escrita na memória no momento do despacho, o mecanismo de reuso atualizará os valores dos registradores identificados no contexto de saída e em seguida enviará a instrução de escrita à estação de reserva e a fila de acesso à memória para manter a integridade das operações de leitura e escrita conforme mostra a **Figura 4.3** a seguir.

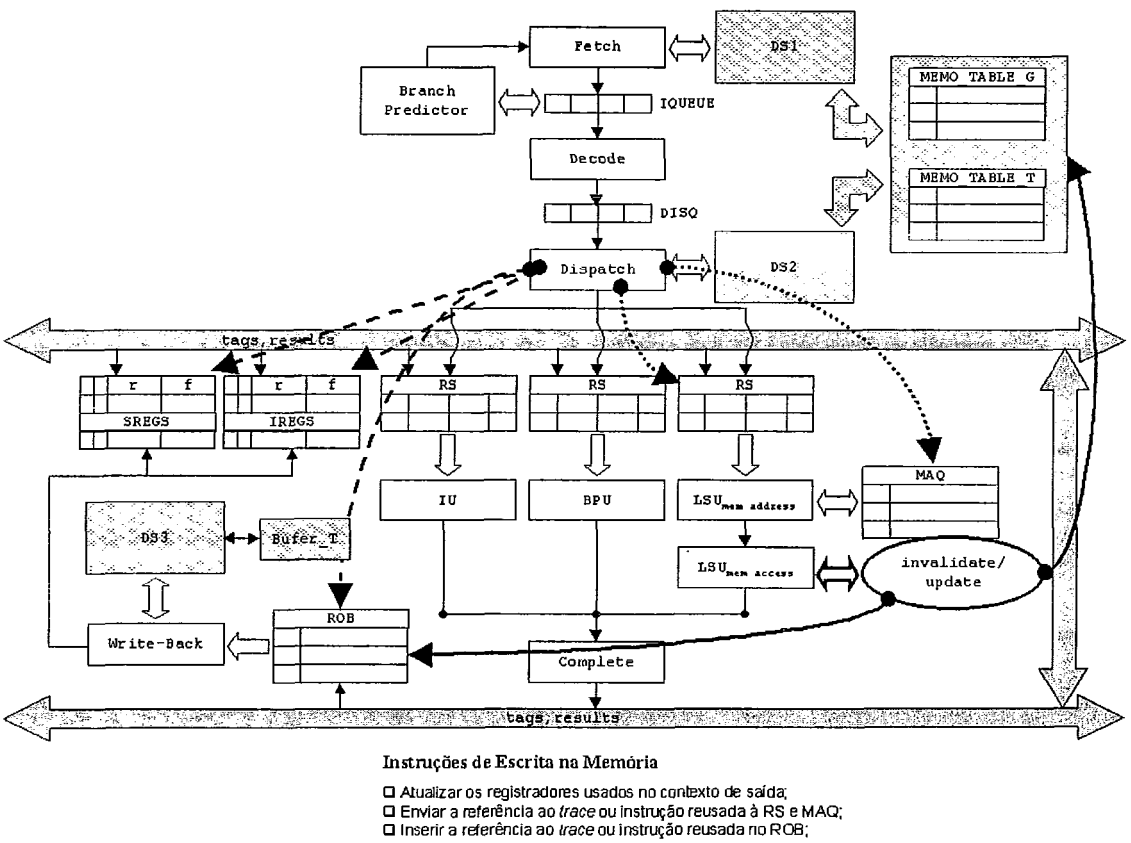


Figura 4.3: Fluxo de dados das instruções ou *traces* reusados contendo *stores*.

Instrução de leitura de memória ou *trace* contendo instruções de leitura, podem seguir dois caminhos diferentes dependendo do estado de consistência dos valores armazenados nas tabelas de reuso. Quando os valores referentes às instruções de leitura são consistentes com os valores contidos na memória a instrução ou *trace* reusado segue o caminho normal do mecanismo, isto é, os registradores identificados no contexto de saída são atualizados junto com o registrador destino da operação de leitura, e o contexto de saída é enviado ao *buffer* de reordenação para finalização. Esta situação é apresentada na **Figura 4.4** pelas linhas tracejadas.

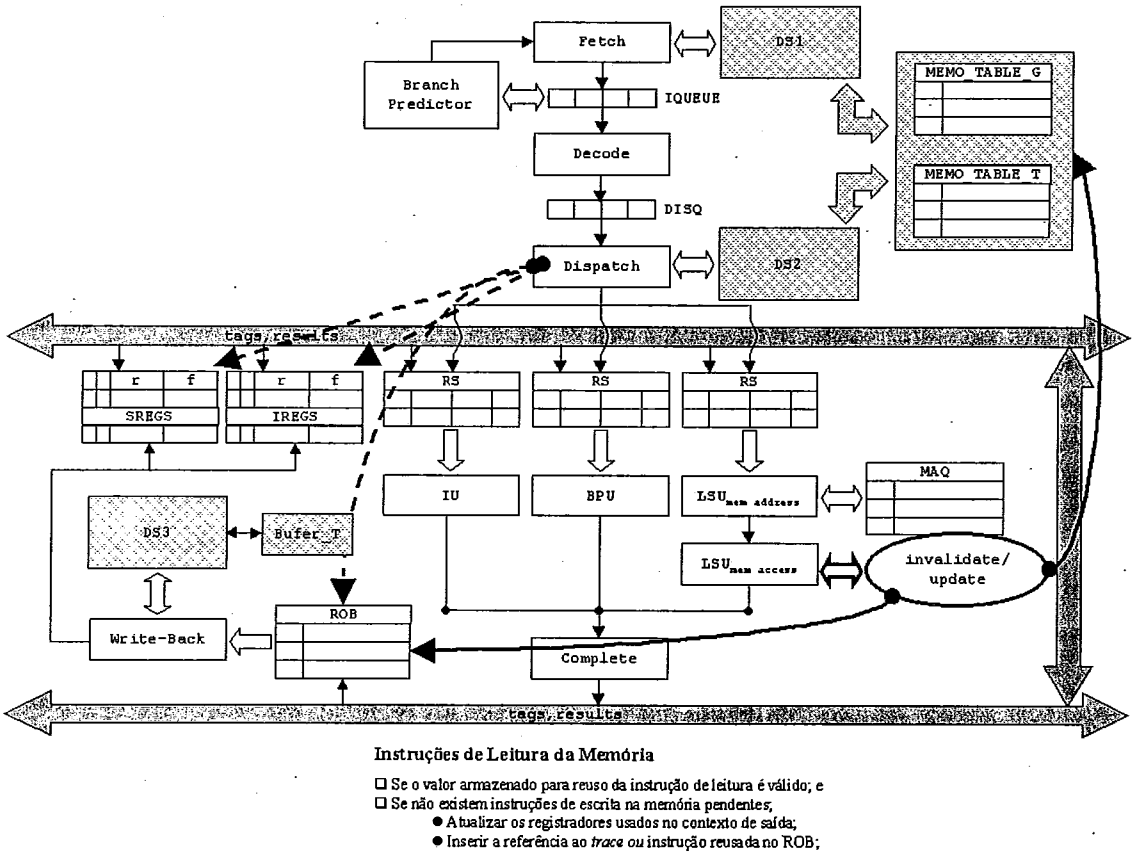


Figura 4.4: Fluxo de dados das instruções ou *traces* reusados contendo *loads*.

Se os valores de retorno estiverem inconsistentes com os valores contidos na memória, a instrução de acesso à memória contida no *trace* é enviada à estação de reserva e a fila de acesso à memória a fim de aguardar a finalização da operação de leitura, conforme é apresentado pelas linha pontilhadas na **Figura 4.5**. Mesmo que instruções de

leitura não reusem o valor do resultado da operação, o endereço de acesso à memória é reusado evitando o calculo do endereço.

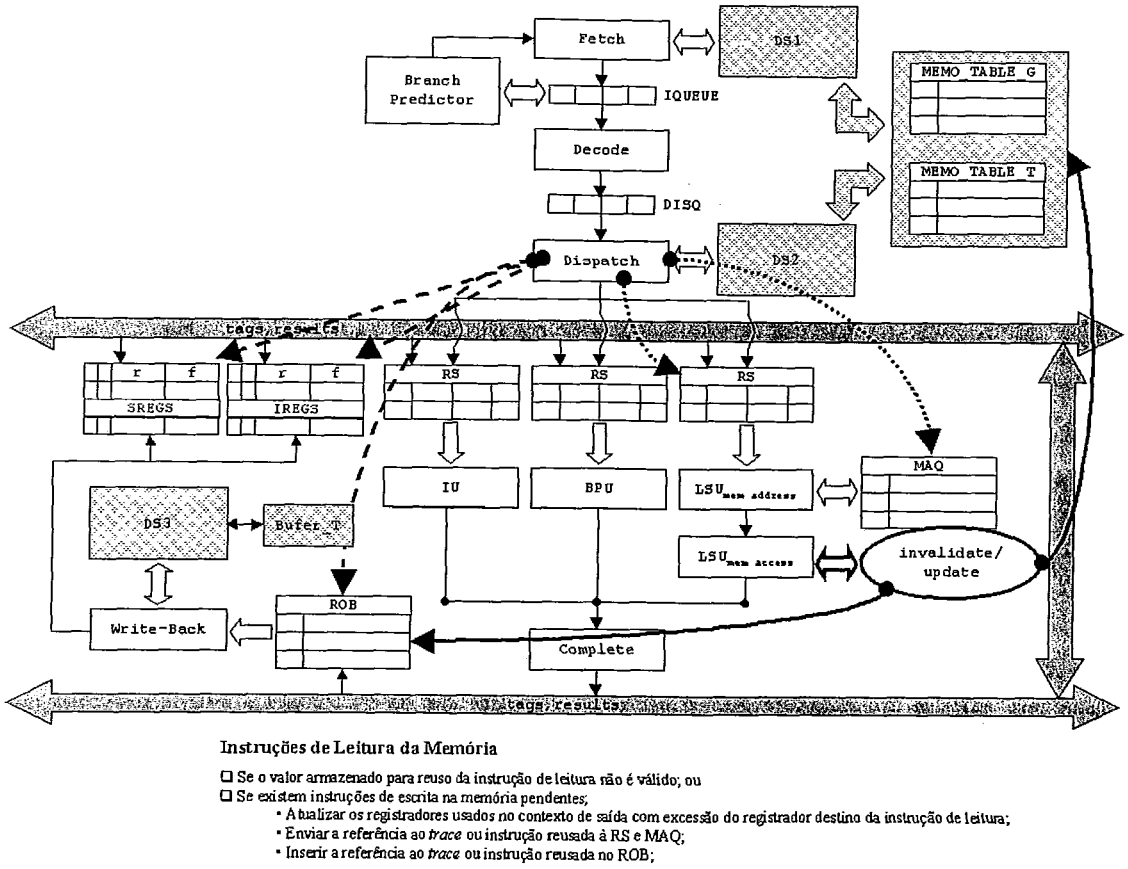


Figura 4.5: Fluxo de dados das instruções ou *traces* reusados contendo *loads*.

Na implementação atual do **DTM** com memória apenas uma instrução de leitura pode ser inserida em um *trace*, e esta instrução precisa ser necessariamente a última instrução. Isto é necessário porque instruções de leitura podem ter seus valores modificados por ações externas ao *trace* e deste modo se estas instruções não estiverem no final da seqüência de instruções o *trace* inteiro terá que ser invalidado. Incluindo instruções de leitura no final dos *traces*, impede que o mesmo seja inteiramente invalidado por operações de escrita que ocorram no mesmo endereço.

A **Listagem 4.1** a seguir mostra uma sequência de instruções, onde as instruções **128** e **12C** são dependentes do valor da instrução de leitura **124**. Assim, se a seqüência de

instruções apresentadas constitui um *trace* e o valor lido da memória não sè repetir, o valor do contexto de saída não representará o resultado correto do *trace*, pois os valores de **R5** e **R7** dependem do valor retornado pela instrução **124** e o *trace* terá que ser invalidado.

```
     instruções                    contexto de saída
100. ADDC R2, R7, #8               R2
104. BNE R2, $124                  R2
124. LD R2 + $1000, R5             R2   (R5)
128. SLL R5, R5, #2                R2   (R5)
12C. SUB R7, R5, #8                R2   (R5)   (R7)
```

Listagem 4.1: Seqüência de instruções de um *trace* com instrução de leitura.

Nas seções seguintes são apresentadas duas variações do **DTM** com memória. A primeira inclui um mecanismo de invalidação das operações de leitura cujos valores são modificados por operações externas, e a outra procura antecipar os valores de operações de memória para evitar invalidações.

4.3 IMPLEMENTAÇÕES DO DTM COM MEMÓRIA

A inclusão de instruções de leitura no **DTM** introduziu também um novo problema. O que devemos fazer quando uma instrução de escrita modifica o valor de uma posição de memória referenciada em um *trace*?

Para resolver este problema temos duas alternativas: Invalidação seletiva dos valores das operações de leitura presentes nas entradas das tabelas de reuso, ou antecipação dos novos valores produzidos por operações de escrita.

4.3.1 IMPLEMENTAÇÃO COM INVALIDAÇÃO DE VALORES

A forma mais simples de adicionar instruções de acesso à memória ao **DTM** é a partir da estrutura original, adicionar um mecanismo de invalidação seletiva das operações de leitura existentes nas entradas das tabelas, conforme pode ser visto na **Figura 4.6**. Nesta figura, um dispositivo de controle foi adicionado à arquitetura original permitindo a anticipação de valores e a invalidação seletiva das tabelas de reuso de acordo com o mecanismo implementado.

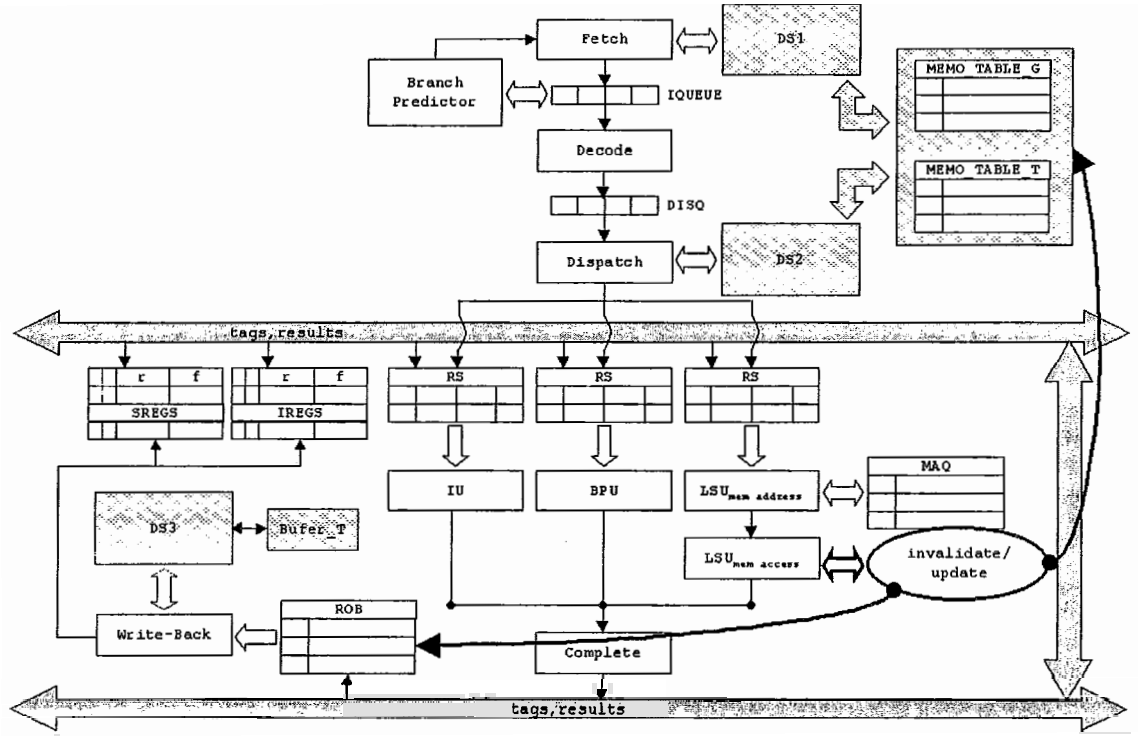


Figura 4.6: Implementação do mecanismo **DTM** com reuso de valores de memória.

Na implementação com invalidação, todas as operações de escrita na memória são interceptadas pelo mecanismo de invalidação que acessa de forma associativa as tabelas de reuso, **MEMO_TABLE_G** e **MEMO_TABLE_T**, e o *buffer* de reordenação, **ROB**, a procura de instruções que lêem do mesmo endereço de memória. Todas as entradas das tabelas de reuso que possuírem operações de leitura na mesma posição de memória que está sendo modificada terão o sinalizador de valor de memória válido para reuso desabilitado. Deste modo, no momento do despacho, quando uma das instâncias

invalidadas for identificada como redundante, o valor deste sinalizador será examinado e em função dele a instrução seguirá o fluxo de dados normal, **Figura 4.4**, ou será enviada a estação de reserva e fila de acesso à memória para execução da operação de leitura conforme é apresentado na **Figura 4.5**.

Na implementação atual, quando uma entrada é invalidada ela se mantém neste estado indefinidamente até que seja removida da tabela de reuso. Isto pode aparentemente degradar o desempenho desta implementação, porém se analisarmos o problema veremos que esta entrada só poderia ser reativada quando fosse efetuada uma operação de escrita na mesma posição de memória atribuindo o mesmo valor. A implementação deste tipo de solução introduz uma complexidade adicional similar a inclusão de um mecanismo de antecipação de valores, e muito provavelmente não contribuirá de forma significativa com o mecanismo. Um caso especial ocorre quando uma instrução de leitura é servida por uma instrução de escrita no mesmo *trace*. Neste caso um *bit* de sinalização é usado para identificar estes *traces* e as invalidações são ignoradas.

4.3.2 SOLUÇÃO COM ANTECIPAÇÃO DE VALORES

A implementação de operações de memória no **DTM** com antecipação de valores tem como objetivo otimizar este mecanismo eliminando a necessidade de invalidação das operações de leitura presentes nos *traces*. Assim como no mecanismo com invalidações, toda operação de escrita em memória é monitorada e o seu endereço comparado com o endereço das operações de leitura presentes nas tabelas de reuso e no **ROB**. Se os endereços coincidirem o valor é repassado às entradas das tabelas de reuso e entradas do **ROB** como pode ser visto na **Figura 4.6**.

Os valores escritos na memória são repassados também ao **ROB** de modo que os valores de retorno produzidos pelas instruções de leitura sejam atualizados com os valores das operações de escrita efetuadas no mesmo endereço. Isto é importante para que um *trace* em formação armazene valores atualizados das operações de leitura, mantendo a consistência dos valores armazenados nas tabelas de retorno. Porém, uma instrução de

leitura servida por operações de escrita em memória presentes no mesmo *trace* não são atualizadas pelo mecanismo.

No momento do despacho, se um *trace* redundante contém uma instrução de escrita na memória, os registradores de saída são atualizados e o contexto de saída do *trace* é enviado ao **ROB**. Além disso, uma referência ao *trace* é inserida na estação de reserva correspondente e na fila de acesso à memória para processamento da operação de escrita, **Figura 4.3**. Se o *trace* possui instruções de leitura os registradores de saída são atualizados e o contexto de saída é enviado ao **ROB** para finalização. Porém se no momento do despacho houverem operações de escrita pendentes de execução a referência ao *trace* é enviada à estação de reserva conforme apresentado na **Figura 4.5**.

4.4 DETALHES DE IMPLEMENTAÇÃO

A implementação de instruções de memória no **DTM** traz alguns custos adicionais ao mecanismo original. Entre estes custos podemos citar a necessidade de adição de mecanismos de invalidação ou antecipação de valores junto aos controles das tabelas de reuso e junto ao *buffer* de reordenação, o que aumenta a complexidade do mecanismo de controle e o tempo de resposta das tabelas.

O aumento do tamanho das entradas das tabelas de reuso também aumentam o custo de implementação. Este aumento no tamanho das entradas é proporcional ao número de operações de acesso à memória permitidas por *trace*. Nos experimentos realizados foram incluídos no máximo uma instrução de acesso à memória por *trace*.

O problema do aumento do tamanho das entradas das tabelas podem ser contornados com a utilização compartilhada da área dedicada ao contexto de entrada e saída com os endereços das instruções de acesso à memória e os valores de retorno. Porém, esta abordagem diminui a quantidade de entradas disponíveis para os operandos do contexto de entrada e saída, influenciando no tamanho médio dos *traces*. Além disso, a inclusão de instruções de acesso à memória aumenta o número de operandos de entrada

dos *traces* e desta forma reduz a freqüência de reuso deles. Para contornar alguns destes problemas uma nova abordagem denominada **DTM*m*** foi desenvolvida e será apresentada no próximo capítulo. Os resultados comparativos entre o **DTM** e as variantes deste mecanismo com reuso de memória são apresentadas no **Capitulo 6.**

Capítulo 5

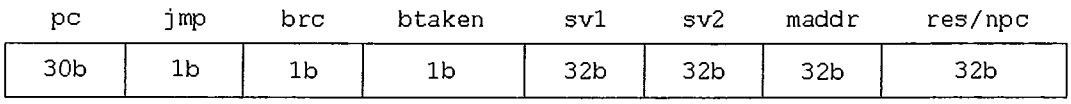
O Mecanismo DTM*m*

Conforme apresentado no capítulo anterior, a implementação de instruções de memória no **DTM** trás alguns custos adicionais ao mecanismo original, tais como o aumento da complexidade do mecanismo de controle das tabelas de reuso e impacto no tempo de resposta das tabelas. Deste modo, a partir das análises das influências exercidas pelas instruções de leitura e de escrita na memória, foi elaborado um novo mecanismo denominado **DTM*m*** que procura resolver alguns dos problemas existentes no mecanismo anterior.

Instruções de Escrita na Memória

Quando uma instrução de escrita na memória possui os mesmos valores de entrada podemos assegurar que o endereço de acesso à memória e o valor que será armazenado é o mesmo da execução anterior. Desta forma, efetuando pequenas modificações na estrutura das tabelas de reuso podemos incluir o reuso de instruções de escrita na memória. Assim, as tabelas de reuso foram modificadas de modo que as entradas pudessem armazenar o endereço de escrita das operações de memória junto com os respectivos valores.

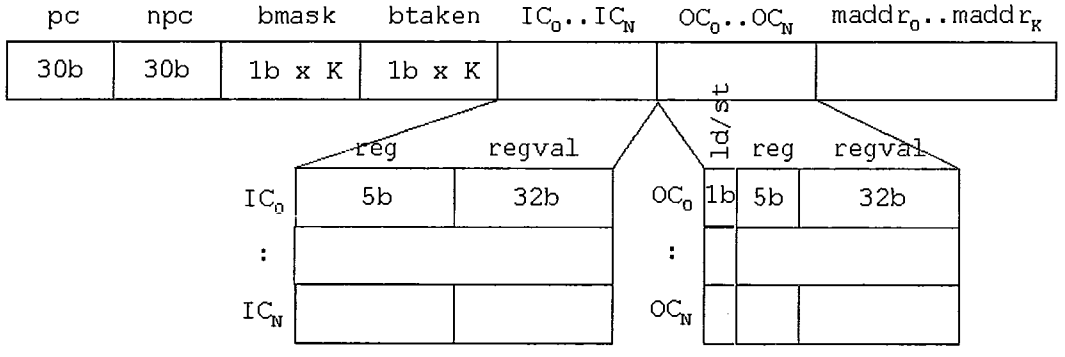
A **Figura 5.1** apresenta as modificações realizadas nas entradas da **MEMO_TABLE_G** e que incluem a adição de um campo para armazenamento do endereço de escrita na memória, **maddr**, além de modificações no campo **res/npc** para incluir o valor a ser armazenado.



pc – endereço da instruçao
jmp – sinalizador de instrução de desvio incondicional
brc – sinalizador de instrução de desvio condicional
btaken – sinalizador de desvio **tomado** ou **nao_tomado**
sv1 – valor do operando fonte
sv2 – valor do operando fonte
maddr – armazena o endereço de acesso à memória
res/npc – resultado da operacao/endereço alvo de desvio/valor à ser armazenado em memória

Figura 5.1: Modificações nas entradas da **MEMO_TABLE_G**.

As modificações realizadas na **MEMO_TABLE_T** incluem a adição de campos para o armazenamento do endereço de acesso à memória $\mathbf{maddr_{(K)}}$. A implementação do **DTM*m*** usa os campos do contexto de saída do *trace* para armazenar o valor a ser escrito na memória, portanto cada entrada no contexto de saída possui um *bit* sinalizador de operação de memória.



pc – endereço da instruçao
npc – endereço da instrução seguinte ao *trace*
bmask – sinalizadores de instruções de desvios
btaken – sinalizadores de desvios **tomado/não tomado**
$IC_0, \ldots, IC_N$ – contexto de entrada do *trace*
reg – endereço do registrador
regval – valor do registrador
$OC_0, \ldots, OC_N$ – contexto de saida do *trace*
reg – endereço do registrador
regval – valor do registrador
ld/st – sinalizador de instrução de acesso à memória
$maddr_0, \ldots, maddr_K$ – endereço de acesso à memória

Figura 5.2: Modificações nas entradas da **MEMO_TABLE_T**.

Quando um *trace* redundante contendo instruções de escrita na memória é despachado, os registradores de saída são atualizados e as instruções de escrita

pertencentes ao *trace* são enviadas a estação de reserva da unidade de acesso à memória, e adicionadas à fila de acesso à memória para processamento.

Instruções de Leitura

A adição de instruções de leitura ao mecanismo difere substancialmente da forma como as instruções de escrita foram tratadas. Este tipo de instrução adiciona parâmetros que são influenciados por operações externas ao trace. Desta forma, não podemos garantir que no momento do reuso de um trace contendo instruções de leitura, os valores contidos na memória serão os mesmos valores armazenados no *trace*. Desta forma um mecanismo de invalidação seletiva ou antecipação de valores precisa ser implementado.

Se um *trace* contém instruções dependentes de alguma operação de leitura da memória contida nele, e alguma instrução de escrita modifica o valor contido no mesmo endereço de memória, não podemos mais garantir que a informação armazenada no contexto de saída do *trace* está correta e desta forma teremos que invalida-lo totalmente. Por isso, no mecanismo de reuso de valores de memória apresentado no capítulo anterior, quando uma instrução de leitura redundante é adicionada ao *trace* em construção, o *trace* é finalizado.

Porém a adição direta de instruções de acesso à memória em um *trace* torna o mecanismo mais complexo. Assim, foi proposta uma nova solução para o problema dos acessos de leitura da memória, que foi separar estas instruções dos *traces*, tratando-as com um esquema de reuso de instruções simples, que armazena em uma estrutura separada denominada **MEMO_TABLE_L** apenas instruções de leitura. Esta solução permite a inclusão de um mecanismo de antecipação de valores com um custo menor já que estatisticamente as instruções de *loads* representam 22% [15] do total de instruções executadas por um programa e exibem 50% a 80% de localidade conforme estudos realizados em [20]. Assim o número de entradas da **MEMO_TABLE_L** pode ser muito menor que o número de entradas da **MEMO_TABLE_G**, reduzindo a complexidade de sua implementação. A **Figura 5.3** apresenta a estrutura de dados da **MEMO_TABLE_L**,

onde o campo **pc** armazena o endereço da instrução e é usado como índice da tabela, **sv1** e **sv2** armazenam os valores dos operandos de entrada, e **maddr** e **rd** armazenam respectivamente o endereço de memória e o valor de retorno da operação de leitura.

pc	sv1	sv2	maddr	mem valid	res
30b	32b	32b	32b	1b	32b.

pc – endereço da instruçao
sv1 – valor do operando fonte
sv2 – valor do operando fonte
maddr – armazena o endereço de acesso à memória
mem valid – sinalizador de valor de leitura de memória válido
res – resultado da oparacao

Figura 5.3: Estrutura das entradas na **MEMO_TABLE_L**.

5.1 DESCRIÇÃO DO MECANISMO

O **DTM*m*** funciona de forma similar aos mecanismos apresentados anteriormente. No estágio de busca, simultaneamente com o acesso à memória para obtenção de novas instruções, é realizado um acesso às tabelas de reuso usando os endereços das instruções como índice. Havendo entradas correspondentes nas tabelas, estas entradas são selecionadas e a instrução é marcada como **redundante**. Caso contrário a instrução é marcada como **não_redundante**.

Instruções que chegam no estágio de despacho marcadas como **redundante** são então avaliadas simultaneamente de acordo com os seguinte critérios. Se existem *traces* selecionados os contextos de entrada deles são avaliados e encontrando redundância o *trace* correspondente será reusado e a instrução marcada como **redundante**. *Traces* reusados atualizam os registradores de saída correspondentes e são enviados ao *buffer* de reordenação. Instruções de escrita na memória pertencentes aos *traces* são enviados a estação de reserva da unidade de execução correspondente e à fila de acesso à memória para execução da operação.

Caso não existam *traces* redundantes é verificado entre as instâncias selecionadas da **MEMO_TABLE_G** se existe alguma instrução redundante. Havendo instruções redundantes os registradores indicados pelos operandos de saída da instrução são atualizados e a instrução inserida no *buffer* de reordenação é marcada como **redundante**. Se a instrução redundante for uma instrução de escrita na memória, ela será enviada a estação de reserva correspondente e a fila de acesso à memória para execução.

Se não houverem instâncias redundantes selecionadas da **MEMO_TABLE_T** e **MEMO_TABLE_G**, é verificado entre as selecionadas da **MEMO_TABLE_L** se existe alguma redundante. Havendo uma instância redundante e estando habilitado o sinalizador de valor de memória válido, os registradores de saída da instrução são atualizados e a instrução é enviada ao *buffer* de reordenação sendo marcada como **redundante**. Se o sinalizador estiver desabilitado ou se houver alguma operação de escrita pendente, a instrução é enviada à estação de reserva da unidade correspondente e a fila de acesso à memória para execução.

No **DTM*m*** todos os *traces* são construídos de maneira similar aos mecanismos apresentados anteriormente, porém instruções de leitura não são inseridas na **MEMO_TABLE_G** nem adicionadas aos *traces*. Após a finalização estas instruções são inseridas na **MEMO_TABLE_L**. Neste mecanismo toda operação de escrita na memória é monitorada e quando um acesso de escrita é efetuado a **MEMO_TABLE_L** é acessada de forma associativa para atualização dos valores de leitura que lêem o mesmo endereço. Invalidações são efetuadas durante operações de entrada e saída para manter a consistência dos dados.

5.2 DETALHES DE IMPLEMENTAÇÃO

A implementação do **DTM*m*** pode ser dividida em duas partes distintas. A primeira cuida da identificação de instruções reusadas e da construção de *traces*. Enquanto a segunda, identifica os *traces* que podem ser reusados e decide entre reusa-los

ou não. A **Figura 5.4** apresenta a implementação do **DTM*m*** em um processador superescalar com *pipeline* de seis estágios.

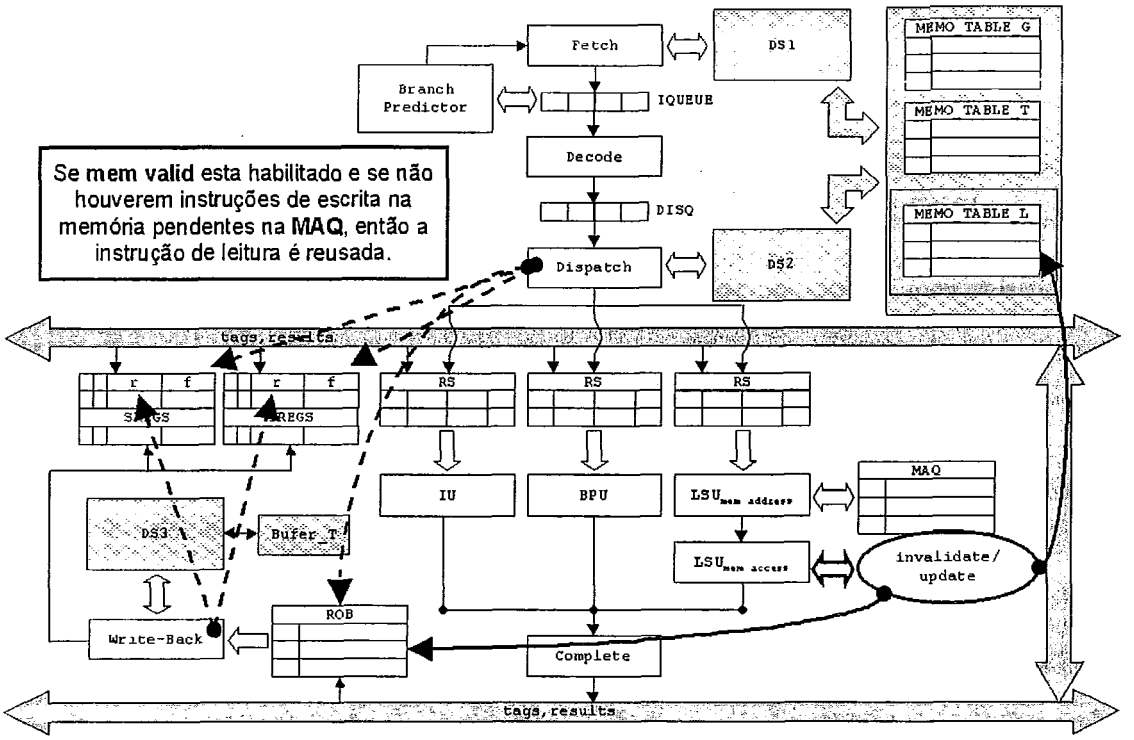


Figura 5.4: Diagrama de blocos de um processador superescalar com **DTM*m***.

Diferente da técnica de reuso apresentada no capítulo anterior, o **DTM*m***, possui um mecanismo de antecipação de valores que atualiza os valores das instruções de leitura presentes no *buffer* de reordenação, **ROB**, e na **MEMO_TABLE_L**. As atualizações das instruções de memória presentes no **ROB** são necessárias pelo fato das instruções de escrita poderem modificar o valor contido num endereço de memória lido por uma instrução de leitura que aguarda para ser inserida na **MEMO_TABLE_L**.

No **DTM*m*** apenas instruções pertencentes ao caminho de execução correto são inseridas nas tabelas de reuso e desta forma não há armazenamento de informações referentes a um caminho especulativo.

Neste capítulo foi apresentado o mecanismo **DTM*m*** e detalhes de sua implementação. No próximo capítulo apresentaremos o ambiente de simulação e os resultados e obtidos.

Capítulo 6

Análise dos Resultados

Os experimentos realizados, utilizaram um subconjunto dos programas do *benchmark* **SPECint '95** apresentado na **Tabela 6.1**. Neste subconjunto não estão incluídos os programas **gcc** e **perl** devido à presença de instruções de ponto flutuante nestes programas. Todos os programas foram compilados utilizando o compilador **gcc-2.5.2** e **glibc-1.0.6**, com otimização **-O**, opção de biblioteca estática, **-static**, e com o recurso de janela de registradores desabilitado **-mflat**.

Quando estes programas são compilados com a opção **-O3** do **gcc-2.5.2** o processo de otimização utiliza os registradores de ponto flutuante do **Sparc v7** para passagem de parâmetros entre funções, e como o simulador **SuperSIM** não implementa estes registradores, os programas do **SPECint '95** foram compilados com a opção **-O**.

Devido ao tempo de simulação, a execução dos programas foi limitada a 100 milhões de instruções, e nesta configuração cerca de 10% do tempo de execução das aplicações são gastos no processamento dos valores de entrada.

Programa	Entrada	Total de Instruções
go	9stone21 (ref)	100.000.000 (não finalizado)
m88ksim	ctl.raw (test)	100.000.000 (não finalizado)
compress	test.in (train)	76.978.452 (finalizado)
li	deriv.lsp (ref)	100.000.000 (não finalizado)
ijpeg	vigo.ppm (ref)	100.000.000 (não finalizado)
vortex	Vortex.in (ref)	100.000.000 (não finalizado)

Tabela 6.1: Parâmetros de entrada usados na execução do **SPECint '95**.

Para execução dos experimentos foi desenvolvido um simulador superescalar, denominado **SuperSIM**, que executa o conjunto de instruções do **Sparc v7**. A descrição do simulador se encontra no **Apêndice A** e a **Tabela 6.2**, a seguir, apresenta os parâmetros de configuração da arquitetura que foram adotados nos experimentos.

	Simplescalar Tool Set	SuperSIM
busca de instruções	4 instruções por ciclo. Apenas um desvio tomado por ciclo. Pode ultrapassar a fronteira da linha de *cache*.	4 instruções por ciclo. Apenas um desvio tomado por ciclo. Não há limite imposto pela *cache*.
cache de instruções	16 Kb, associativa - 2 por conjunto, 32 bytes por linha, latência de 6 ciclos para *miss* no *cache* L1 e 20 ciclos para *miss* no *cache* L2.	Não implementado, considera acerto na *cache* de 100%.
preditor de desvios	Bimodal, 2k entradas, pode predizer vários desvios simultaneamente.	Bimodal, 1k entradas, pode predizer vários desvios simultaneamente.
mecanismo de execução especulativa	Execução de até quatro instruções por ciclo fora de ordem, *buffer* de reordenação com 16 entradas e fila de acesso à memória com 8. Loads são executados após serem conhecidos todos os endereços de *stores* precedentes. *Loads* são servidos por *stores* que acessam o mesmo endereço se ambos estiverem na fila de acesso à memória.	Execução de até quatro instruções por ciclo fora de ordem, *buffer* de reordenação com 16 entradas e fila de acesso à memória com 16. Loads são executados após todos os *stores* precedentes terem sido executados. Instruções de *loads* não são servidas por *stores*.
registradores arquiteturais	32 registradores de inteiros, 32 registradores de ponto flutuante, registradores hi, lo e fcc.	Janela de registradores com 520 registradores de inteiros e 32 registradores ativos por janela, sendo 8 registradores fixos e 32 x 24 registradores sobrepostos. 2 registradores

		especiais PSR e Y.
unidades funcionais	4 ULAs de inteiros, 2 unidades *load/store*, 4 *adders* de ponto flutuante, 1 mult/div inteiro, 1 mult/div ponto flutuante.	3 ULAS que resolvem operações com inteiros incluindo multiplicação. 2 unidades de desvios e 2 unidades de *loads/stores*.
latência das unidades funcionais	ULA-inteiros/1, *load/store*/1, mult int/3, int div/20, fp *adders*/2, fp mult/4, fp div/12, fp sqrt/24.	Todas as instruções de inteiro possuem latência 1. Instruções de *load/store* possuem latência 2.
cache de dados	16Kb, associativo-2 por conjunto, 32 bytes por linha, latência de 6 ciclos para *miss* no *cache* L1 e 20 ciclos para *miss* no *cache* L2.	Não implementado, considerado acesso 100% na *cache*.

Tabela 6.2: Parâmetros de configuração da arquitetura dos simuladores.

Na **Tabela 6.2** podemos observar que algumas características arquiteturais, tais como *cache* de instruções e dados, e latência de instruções não foram implementadas no simulador **SuperSIM** para simplificação do projeto.

A **Tabela 6.3** apresenta os parâmetros de configuração do **DTM**. Podemos observar nesta tabela que o número de registradores nos contextos de entrada e saída dos *traces* foram ampliados devido à existência de dois registradores especiais, **psr** e **y**. O registrador **psr** armazena os sinalizadores de estado da arquitetura e o registrador **y** é usado por operações de multiplicação parcial. O conjunto de instruções do **Sparc v7** possui operações que atualizam mais de um registrador e por este motivo o número de campos destinados aos contextos de entrada e saída dos *traces* foram ampliados.

	DTMmips	DTMsparc
contexto de entrada	6 entradas	7 entradas
contexto de saída	6 entradas	7 entradas
tamanho max dos traces	16 instruções	Ilimitado
Número max de desvios	10 desvios	10 desvios
heurística	Repetição de instruções simples.	Repetição de instruções simples.
conjunto de seleção	Instruções aritméticas, lógicas, desvios, chamadas e retorno de sub-rotina, e cálculo do endereço de acesso à memória.	Instruções aritméticas, lógicas, desvios, chamada e retorno de sub-rotina, e cálculo do endereço e valor de acesso à memória.
política de atualização das tabelas de reuso	LRU	LRU
tabelas de reuso	MEMO_TABLE_G - 4672 entradas, associativa. MEMO_TABLE_T - 512 entradas, associativa.	MEMO_TABLE_G - 4672 entradas, associativa. MEMO_TABLE_T - 512 entradas, associativa. MEMO_TABLE_L - 512 entradas, associativa.

Tabela 6.3: Parâmetros de configuração do **DTM** e extensões.

6.1 MÉTRICAS UTILIZADAS

De forma similar aos experimentos realizados em [8], as métricas básicas adotadas neste experimento foram percentual de reuso e de aceleração.

- Percentual de reuso é representado por:

$$\% \text{ reuso} = ir / itot \qquad \textbf{(Equação 6.1)}$$

Onde ir - número de instruções reusadas
itot - número de instruções executadas

- Percentual de aceleração é representada por:

$$\text{aceleração} = IPCt / IPCbase \qquad \textbf{(Equação 6.2)}$$

Onde IPCt - instruções executadas por ciclo com o mecanismo.
IPCbase - instruções executadas por ciclo da arquitetura base.

Os valores médios apresentados neste capítulo são fornecidos por médias aritiméticas e harmônicas dos dados conforme as expressões à seguir:

- Média aritimética é obtida da expressão:

$$AM = \left(\sum_{i=0}^{n} S_i \right) / n \qquad \textbf{(Equação 6.3)}$$

Onde n - total de valores computados
S_i - valores computados

- Média harmônica é obtida da expressão:

$$HM = n \left(\sum_{i=0}^{n} 1/S_i \right)^{-1} \qquad \textbf{(Equação 6.4)}$$

Onde n - total de valores computados
S_i - valores computados

6.2 DISTRIBUIÇÃO DAS INSTRUÇÕES NOS PROGRAMAS

A **Tabela 6.4** e o gráfico apresentado na **Figura 6.1** mostram a freqüência de execução dos diferentes tipos de instruções nos programas analisados. Podemos observar, que a freqüência de instruções de *loads* e *stores* juntas representam em média 34% das instruções executadas por um programa, e a freqüência de instruções de leitura da memória representam 22%. Estes resultados são similares aos resultados obtidos em [15] com a arquitetura **CISC** do **Intel 80x86**, e que esta reproduzido na **Tabela 1.1**.

Podemos observar, na **Tabela 6.4**, que a freqüência de instruções de *save* e *restore* foram muito baixas, e este resultado se deve principalmente ao tipo de compilação adotada. Os programas analisados foram compilando com o parâmetro – **mflat**, que desabilita o mecanismo de janela de registradores e reduz significativamente o número de instruções de *save* e *restore*, e a redução do número de instruções que manipulam janelas de registradores, induz o aumento no número de instruções de acesso à memória na arquitetura **Sparc v7**.

	go	m88ksim	Compress	Li	ijpeg	Vortex	AM
Call	431747	733152	939892	1647111	2205754	1119034	1179448
Bicc	8509903	12592476	8031639	12185567	11028625	11947490	10715950
Jmpl	451801	814256	940050	2002746	2206465	1134519	1258306
Ticc	12	65	12	203	546	4224	844
Load	22677213	17947340	12084545	25117843	24989669	24552343	21228159
Store	6092792	7958462	8263676	13822452	15428577	16871675	11406272
arithmetic	12701672	18469412	15563675	17263697	19833903	19553774	17231022
Logic	29518067	22011463	14355316	7829303	11038960	12143797	16149484
Mult	376865	6691	10	125	33251	210884	104638
Save	277	568	16672	943	567	21097	6687
restore	277	568	16671	943	567	21097	6687
Sethi	19142940	19463133	16766290	20129041	13230038	12370162	16850267
Others	96434	2414	4	26	3078	49904	25310

Tabela 6.4: Freqüência dos diferentes tipos de instruções nos programas.

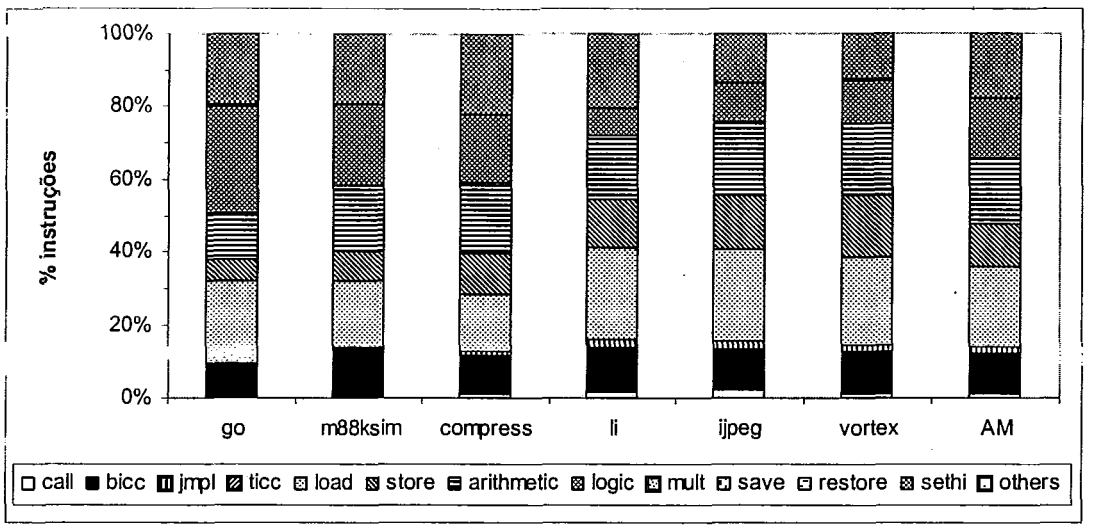


Figura 6.1: Freqüência dos diferentes tipos de instruções nos programas.

Na **Figura 6.1**, observamos que as instruções *sethi*, que atribuem um valor constante aos *bits* de maior ordem de um registrador, também foram bastante representativas para as aplicações avaliadas, sendo que em média 19% das instruções executadas, foram deste tipo. Estas instruções são freqüentemente usadas como instrução de *nop*, e para o posicionamento dos registradores nos endereços bases de estruturas de dados.

6.3 RESULTADOS DA IMPLEMENTAÇÃO DO DTM NO SPARC

A **Figura 6.2** compara a aceleração obtida com o **DTM** nos experimentos realizados por [8], onde **DTM*mips*** representa a implementação deste mecanismo com a configuração de *hardware* apresentada nas **Tabelas 6.2** e **6.3**, e **DTM*mips_perf*** representa este mecanismo com a mesma configuração de *hardware* e *cache* perfeito. **DTM*sparc*** representa a implementação do **DTM** que foi realizada no simulador **SuperSIM**, e **DTM*sparc_0k*** representa a mesma implementação, considerando apenas o reuso de instruções simples.

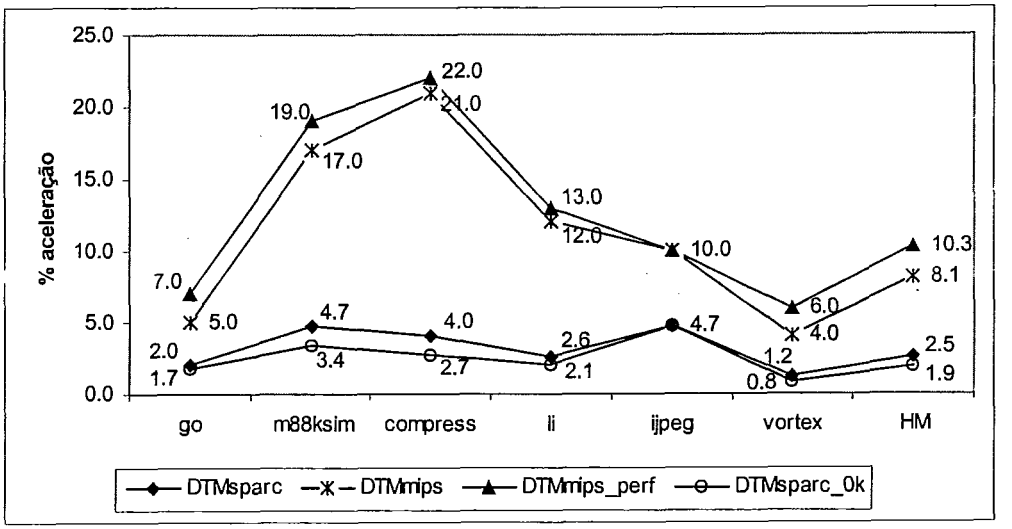


Figura 6.2: Comparação entre a aceleração de **DTMmips** e **DTMsparc**.

Analisando a **Figura 6.2** observamos que a diferença de desempenho obtida com o **DTM*mips*** em [8] foi de 5,6% em relação ao **DTM*sparc***, quando comparamos as médias harmônicas dos dois experimentos. A figura também mostra que o resultado do **DTM*sparc*** foi em média 0,6% acima do resultado obtido considerando apenas o reuso de instruções simples.

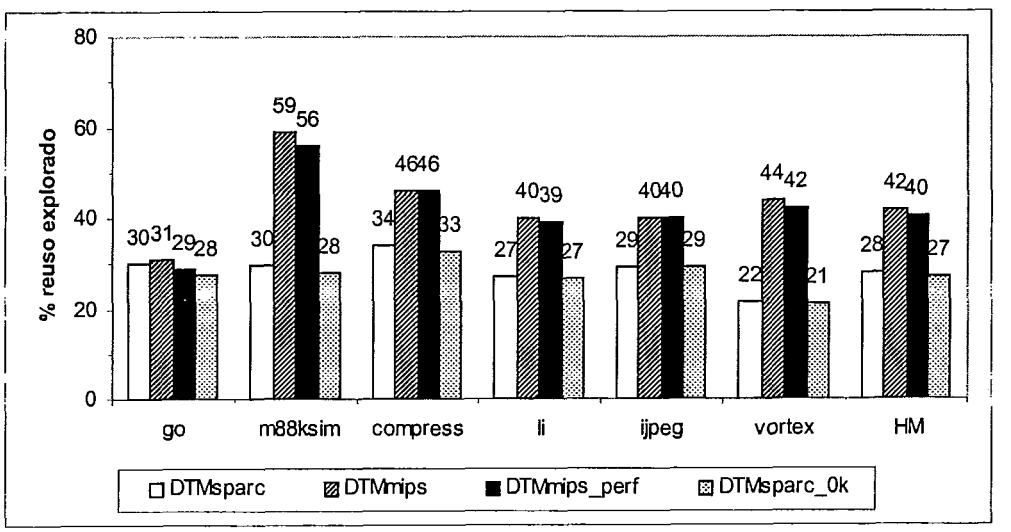


Figura 6.3: Reuso explorado com DTMmips x DTMsparc.

Analisando o gráfico da **Figura 6.3**, que mostra o percentual de reuso explorado com o **DTM*mips*** e **DTM*sparc***, observamos que o reuso explorado com o **DTM*mips*** foi superior em 14% ao resultado obtido com o **DTM*sparc***.

A **Tabela 6.5** e a **Figura 6.4** apresentam a distribuição dos diferentes motivos de finalização dos *traces* que contribuíram com o reuso explorado pelo **DTM*sparc***. Onde, **overflow** representa os *traces* finalizados por *overflow* dos contextos de entrada ou de saída. Os *traces* finalizados por instruções em *delay slot* são denominados por **delayinst**. *Traces* finalizados por instruções não válidas são representados por **nvalid**, e instruções não redundantes por **nredundant**. Os *traces* finalizados por *traces* reusados, são identificados por **trace** na figura, e **loadstore** representa *traces* finalizados por instruções de acesso à memória.

Na **Figura 6.4**, observamos que a ocorrência de *traces* finalizados por *delay instructions* foi significativa, chegando a representar 74% dos *traces* reusados no **m88ksim**. Este resultado contribui significativamente com o baixo resultado alcançado pelo **DTM*sparc***. Além disso, fatores como as latências das instruções e o tamanho da tabela de predição de desvios contribuíram com o melhor resultado apresentado pelo **DTM*mips*.**

Podemos observar nas **Figuras 6.2** e **6.3** que os resultados obtidos em [8] para o **DTM*mips_perf***, foi ligeiramente superior aos resultados obtidos com o **DTM*mips***. Isto foi motivado pelo fato dos *traces* finalizados por instrução de *load* reusarem o valor do endereço de acesso à memória e, estes acessos à memória podem, no **DTM*mips***, resultar em *miss* da *cache*, introduzindo uma penalidade que encobre o ganho obtido pelo reuso do *trace*. Considerando *cache* perfeito esta penalidade deixa de ocorrer e o desempenho do mecanismo melhora [8].

	overflow	Delayinst	nvalid	nredundant	Trace	loadstore
Go	0.00	6.22	0.00	83.07	0.39	10.32
M88ksim	0.00	74.30	0.00	22.97	0.00	2.73
Compress	0.00	13.10	0.00	59.99	0.00	26.91
Li	0.00	24.15	0.00	57.43	0.00	18.42
Ijpeg	0.00	33.33	0.00	66.67	0.00	0.00
Vortex	0.00	19.68	0.00	75.12	0.04	5.16
AM	0.00	30.86	0.00	56.02	0.10	13.01

Tabela 6.5: Motivos de finalização dos *traces* com **DTM*sparc***.

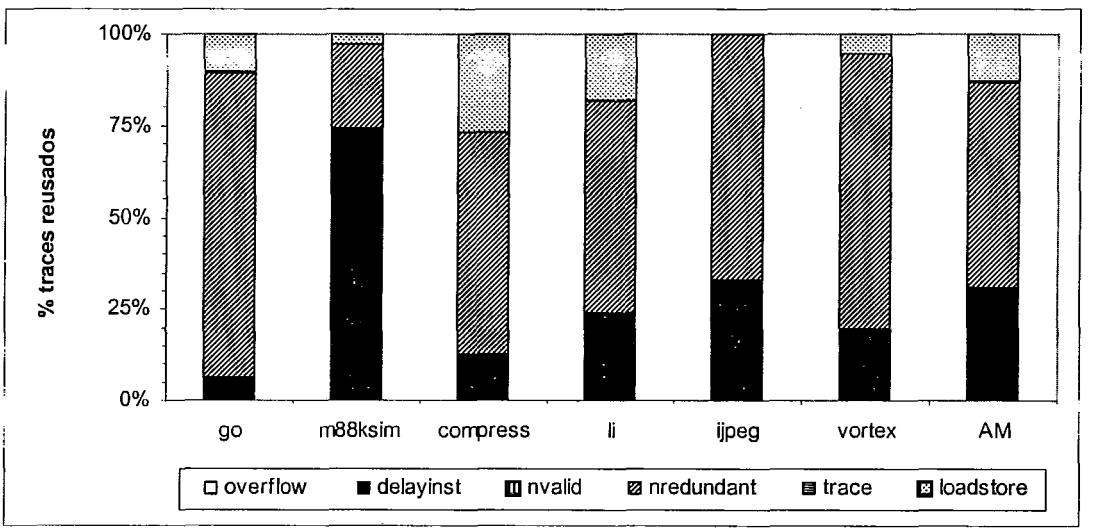


Figura 6.4: Motivos de finalização dos *traces* com **DTM*sparc***.

Os resultados obtidos com o **compress** também apresentaram diferenças significativas de desempenho embora a diferença entre o reuso explorado pelas duas implementações não tenha sido tão grande. Na **Figura 6.4** observamos que no **compress** cerca de 27% dos *traces* reusados foram finalizados por instruções de acesso à memória e o simulador **SuperSIM** usa uma política de acesso à memória mais rígida, que contribuiu para o rendimento inferior do **DTM** nesta arquitetura.

A **Tabela 6.6** à seguir apresenta o motivo de finalização dos *traces* não formados, isto é, dos *traces* em construção que contém apenas uma instrução e que por algum motivo foram finalizados e descartados. Nesta tabela, observamos que o principal

motivo de finalização de *traces* não construídos é a ocorrência de instruções não redundantes. Em seguida, observamos que instruções ocupando posição de *delay slot* é o segundo motivo mais freqüente. Isto significa que muitos *traces* em formação conténdo uma instrução de desvio, foram descartados.

Na tabela **Tabela 6.6**, observamos que instruções de leitura e escrita na memória também representam um motivo freqüênte de finalização de *traces* não formados, e a implementação do **DTM** com reuso de valores de instruções de acesso à memória, pode reduzir bastante a ocorrência de *traces* não formados por este motivo.

A **Tabela 6.6** mostra uma quantidade significativa de *traces* não formados finalizados por *traces* reusados. Estes *traces* não formados poderiam ser facilmente concatenados com os *traces* reusados produzindo *traces* maiores.

	delayinst	nredundant	loadstore	Trace	nvalid
Go	2298042	7047053	740022	6131	1
M88ksim	3009151	7726553	374041	6974	9
Compress	3910282	6593024	1035892	139068	4
Li	6494119	3513903	582683	760	4
Ijpeg	8079268	5007030	589384	0	3
Vortex	2556571	6680095	768659	1419	837
AM	4391239	6094610	681780	25725	143

Tabela 6.6: Motivos de finalização dos *traces* não formados no **DTM*sparc***.

Outro fator significativo para o desempenho favorável do **DTM*mips***, foi o número de instruções de desvios encapsuladas nos *traces*. A **Figuras 6.5**, mostra o percentual dos *traces* reusados pelo número de instruções de desvios contidas neles. Nesta figura, observamos que no **DTM*sparc*** 69% dos *traces* reusados não contém instruções de desvios, e 31% deles possum apenas um desvio. Observamos também que o número de *traces* reusados com mais de um desvio é inferior à 1% nesta implementação.

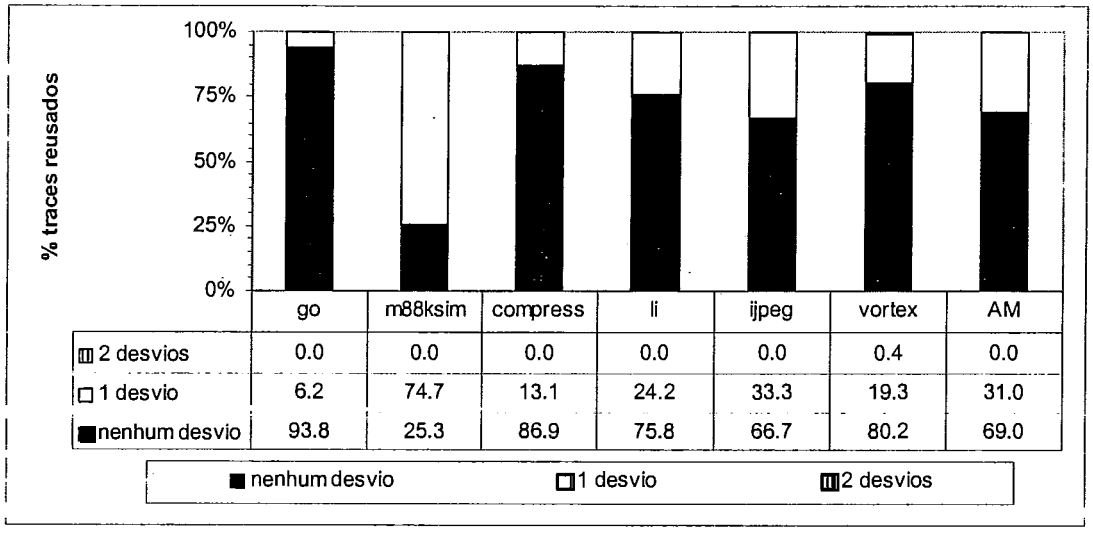


Figura 6.5: Freqüência dos *traces* por número de desvios no **DTM*sparc***.

Na **Figura 6.6**, observamos no **DTM*mips*** que 39% dos *traces* não possuem desvios, isto é, aproximadamente metade do número encontrado na implantação do **DTM*sparc***. Além disso, 38.5% dos *traces* possuem um desvio, e 22% deles possuem mais de um desvio.

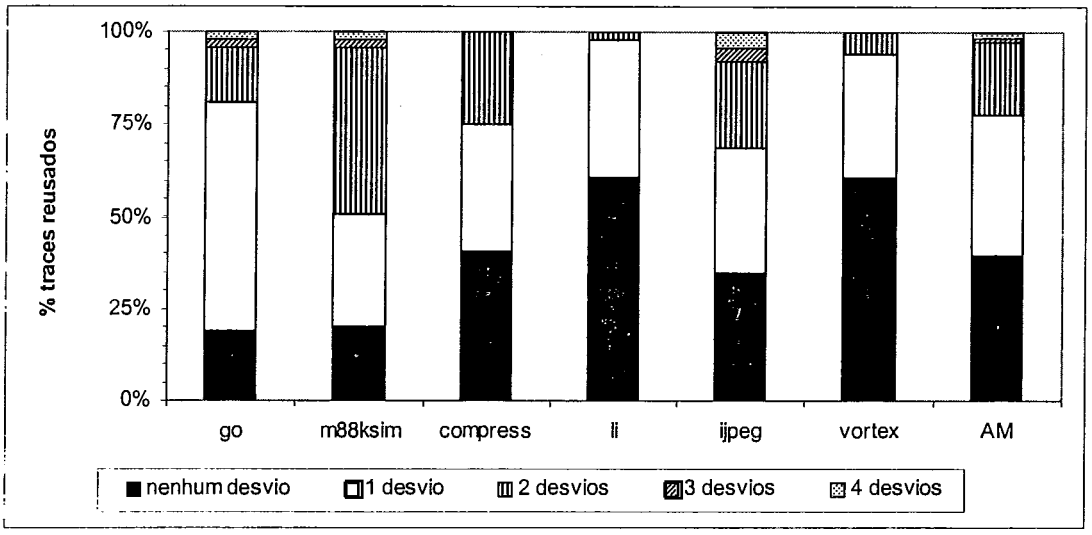


Figura 6.6: Freqüência dos *traces* por número de desvios no **DTM*mips***.

Quando um *trace* reusado possui mais de um desvio, as penalidades que poderiam ocorrer em função de uma previsão incorreta de um dos desvios do *trace* é eliminada em função da certeza proporcionada pelo reuso do *trace*. Deste modo, um *trace* com muitos desvios pode produzir ganhos maiores de aceleração em uma arquitetura que implementa reuso de *traces*, pois eliminam as penalidades provocadas por falha do preditor.

Nesta seção foi avaliado a implementação do **DTM** no simulador **SuperSIM**, comparando os resultados obtidos com os encontrados nos experimentos conduzidos em [8]. Na próxima seção iniciaremos a análise dos resultados obtidos com a implementação de reuso de valores de instruções de acesso à memória no **DTM**.

6.4 RESULTADOS DAS DIFERENTES IMPLEMENTAÇÕES DO DTM COM REUSO DE INSTRUÇÕES DE MEMÓRIA

Os gráficos das **Figuras 6.7** e **6.8** apresentam respectivamente os resultados de aceleração e percentual de reuso obtidos pelas diferentes implementações do **DTM** na arquitetura **Sparc v7**, onde **DTM** representa os resultados obtidos com a implementação do mecanismo original, **DTM*inv*** representa a implementação do **DTM** com reuso de instruções de acesso à memória com invalidação, **DTM*upd*** identifica o mecanismo que implementa reuso de *traces* com instruções de acesso à memória e antecipação de valores de *loads*, e **DTM*m*** representa a extensão do **DTM** com reuso de valores de operações de acesso à memória, implementado com uma tabela de load separada.

Podemos observar nos gráficos das **Figuras 6.7** e **6.8**, que a aceleração média dos mecanismos que implementam reuso de *traces* com antecipação de valores de memória, foram em média 4,1% e 2,9% acima do resultado obtido com o **DTM**. Porém, o reuso explorado por estes mecanismos tiveram uma variação percentual de 1%. Além disso, a aceleração média obtida pelo mecanismo com invalidação de valores de memória foi igual a obtida com o **DTM**.

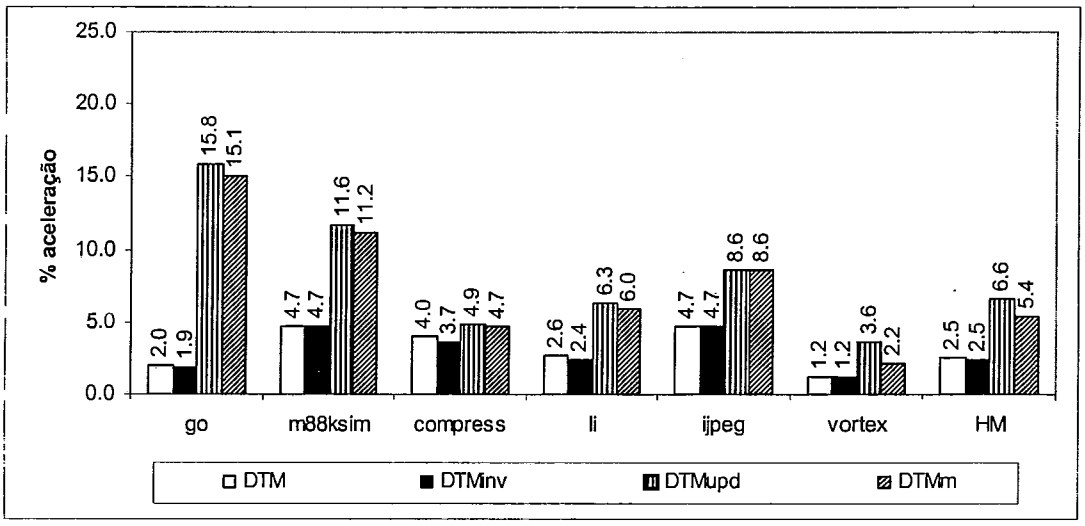


Figura 6.7: Aceleração obtida com as implementações do **DTM** no **Sparc v7**.

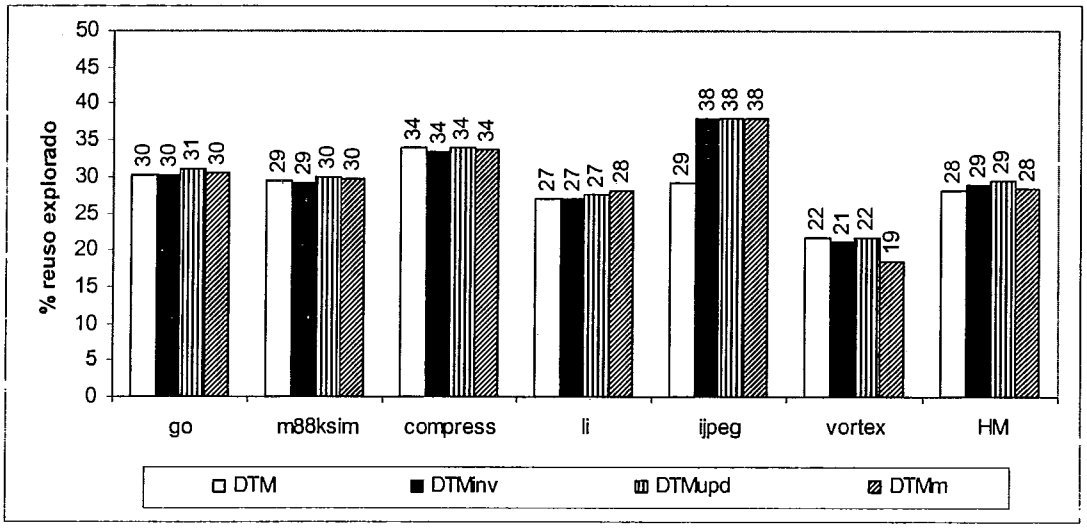


Figura 6.8: Reuso explorado com as implementações do **DTM** no **Sparc v7**.

Estes resultados demonstram que o reuso de instruções de acesso à memória com antecipação de valores produz ganhos significativos de aceleração, melhorando a qualidade dos *traces* e instruções reusadas pelo mecanismo.

6.4.1 ANALISANDO OS RESULTADOS DO DTM*inv*

Conforme podemos observar nas **Figuras 6.7** e **6.8**, o desempenho alcançado pelo **DTM*inv*** foi equivalente ao desempenho obtido com o **DTM** para a maioria das aplicações analisadas. Isto ocorreu pelo fato do **DTM*inv*** explorar pouco o reuso de valores de *loads* devido as freqüentes invalidações nas tabelas de reuso.

Foi observado também um aumento no número de *traces* reusados compostos por duas instruções, uma instrução simples e uma instrução de escrita na memória, **Tabela 6.7**, e o reuso destes *traces* é similar ao reuso de instruções simples pois a instrução de escrita não retorna valor após a computação e o seu reuso não traz ganhos significativos ao mecanismo. Além disso, estes *traces* ocupam entradas na **MEMO_TABLE_T** que poderiam ser utilizadas por *traces* de melhor qualidade que proporcionariam ganhos maiores ao **DTM*inv***.

Instruções de *stores* também oferecem pouco ganho em relação ao reuso de instruções de *loads*, pois os *traces* que reusam tais instruções são enviados às estações de reserva da unidade de execução e a fila de acesso à memória ocupando tais recursos.

	2 instruções		3 instruções		4 instruções	
	store	load	store	load	Store	load
go	82272	0	58426	333015	0	3624
M88ksim	39133	0	51465	89056	11995	2
compress	24619	0	13830	600920	0	226179
li	141802	0	142255	295577	6	86
ijpeg	0	0	2199272	0	0	0
vortex	166399	0	51416	25008	0	5473
AM	47971	0	410875	219761	2000	38315

Tabela 6.7: Freqüência dos *traces* com *loads* e *stores* pelo tamanho.

A adição de instruções de acesso à memória também aumentou significativamente a freqüência de *traces* finalizados por *overflow* do contexto de entrada e saída, o que impediu o reuso de *traces* com maior número de instruções, conforme

podemos observar na **Tabela 6.8** e no gráfico correspondente apresentado na **Figura 6.9** que mostram a distribuição dos motivos de finalização dos *traces* reusados pelo **DTMinv**.

	overflow	delayinst	Nvalid	nredundant	trace	loadstore
Go	1.97	6.26	0.00	83.61	0.37	9.76
M88ksim	0.32	72.75	0.00	23.48	0.06	3.71
compress	0.45	12.94	0.00	60.44	0.00	26.62
Li	5.71	23.44	0.00	60.80	0.00	15.76
Ijpeg	0.00	0.00	0.00	0.00	0.00	100.00
vortex	0.95	15.37	0.00	78.42	0.16	6.05
AM	1.45	25.67	0.00	49.35	0.11	24.87

Tabela 6.8: Motivos de finalização dos *traces* com **DTM*inv*** no **Sparc v7**.

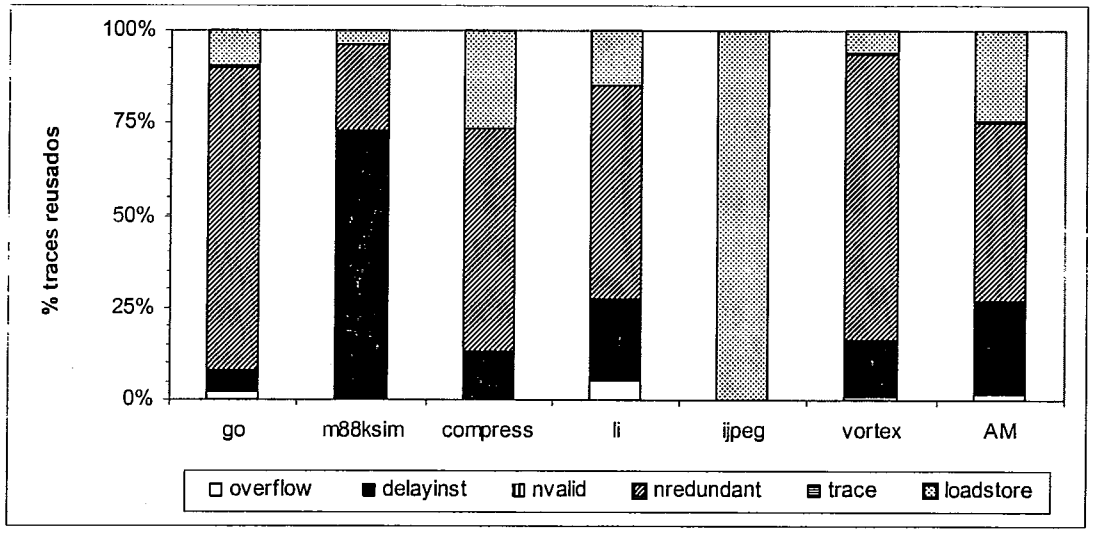


Figura 6.9: Motivos de finalização dos *traces* com **DTM*inv*** no **Sparc v7**.

6.4.2 ANALISANDO OS RESULTADOS DO DTM*upd* E DTM*m*

Na **Figuras 6.7**, observamos que os mecanismos que implementam reuso de instruções de acesso à memória com antecipação de valores de *loads* obtiveram ganhos de aceleração significativos em relação ao **DTM** e **DTM*inv***. Porém, o reuso explorado por estas implementações mantiveram resultados equivalentes, conforme mostra a **Figura 6.8**. Existem alguns fatores que contribuem para o aumento significativo no

desempenho destes mecanismos, e entre eles podemos destacar as limitações impostas pelo mecanismo de acesso à memória usado no simulador, e o reuso de instruções de *loads* que são freqüentemente empregados na inicialização de uma computação, e que quando reusadas, antecipam seus resultados e contribuem significativamente com o mecanismo.

Na **Figura 6.4** observamos que os *traces* finalizados por instruções de *loads* e *stores* são bastante representativos na implementação do **DTM** embora este mecanismo não faça reuso do valor de retorno destas operações. Assim, a política rígida de acesso à memória implementada pelo simulador **SuperSIM** contribui significativamente para o salto de desempenho obtido com a inclusão de reuso de valores de instruções de acesso à memória, que aliviou a sobrecarga na fila de acesso à memória. Outro fator representativo foi à compilação aplicada nos programas de *benchmark* que introduziu grande quantidade de instruções de *load* e *store*.

Analisando a **Figura 6.10** observamos que o percentual de *traces* reusados servidos por *loads* redundantes representam em média 31% para o **DTM*upd*** e 23% no **DTM*m***, e estes resultados contribuem bastante para a aceleração obtida por estes mecanismos.

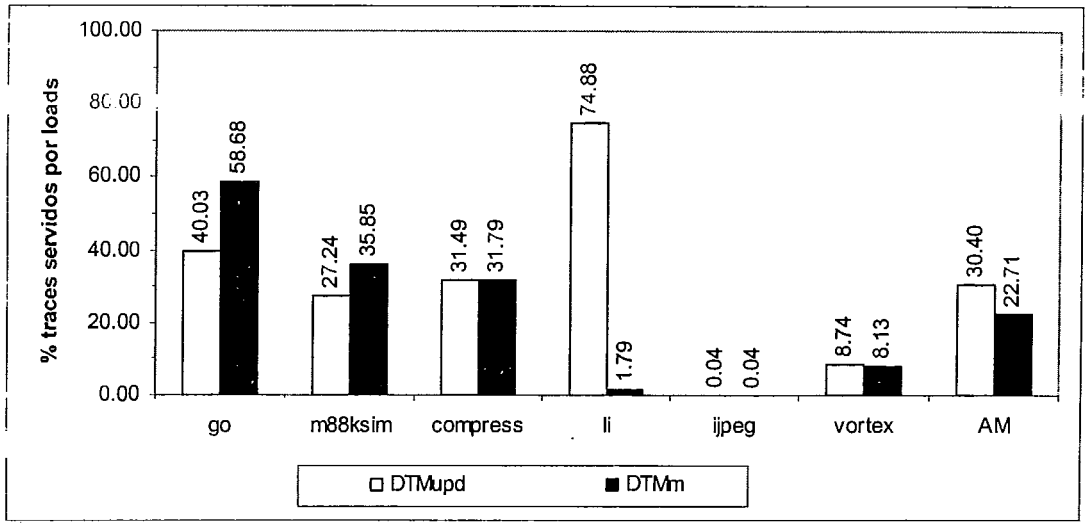


Figura 6.10: Percentual dos *traces* servidos por *loads* reusados.

6.4.3 COMPARANDO OS RESULTADOS DE DTM*upd* E DTM*m*

Analisando a **Figura 6.7** observamos que **DTM*upd*** e **DTM*m*** obtiveram a mesma aceleração embora fosse esperado que o reuso de *loads* contidos em *traces* de instruções fornecesse ganhos maiores. Podemos explicar tais resultados de aceleração se observarmos o gráfico da **Figuras 6.11**, que apresenta o percentual de *loads* reusados com *traces* e como instruções simples.

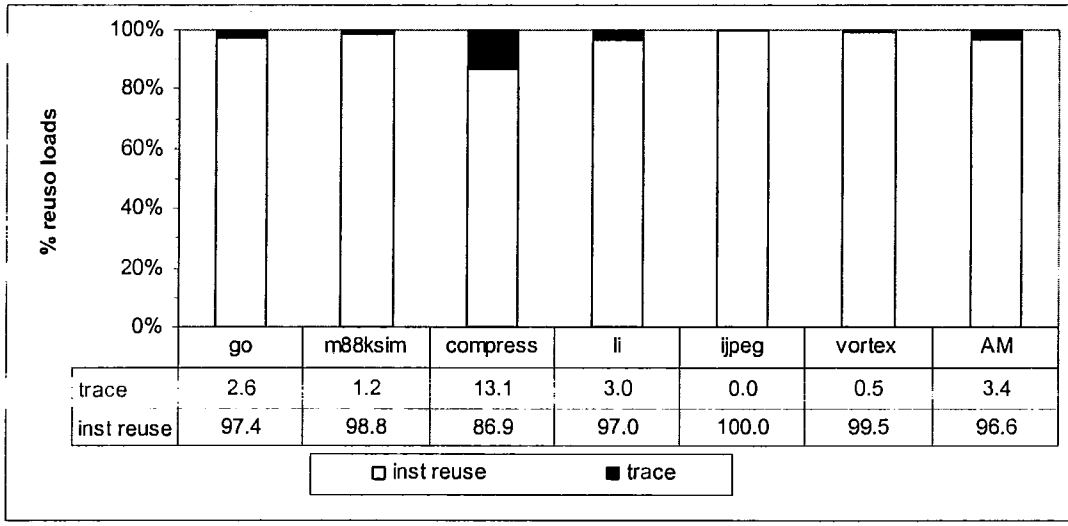


Figura 6.11: Percentual dos *loads* reusados por *traces* e instruções no **DTM*upd***.

Nesta figura podemos observar que o reuso de *loads* reusados como instruções simples representa cerca de 97% do total de reuso de *loads* encontrado nos programas de *benchmark*. Além disso, os *traces* contém no máximo um *load* e esta limitação também influencia nos resultados obtidos.

Também a implementação do **DTM*m*** adiciona uma tabela de reuso de *loads* com 512 entradas, ampliando o potencial deste mecanismo em reusar instruções simples já que as 4672 entradas da **MEMO_TABLE_G** foram mantidas. Este aumento no potencial de reuso de instruções simples também refletiu diretamente na aceleração obtida por este mecanismo.

6.5 RESULTADOS OBTIDOS COM O REUSO DE CADEIAS DE INSTRUÇÕES E *TRACES* DEPENDENTES EM UM MESMO CICLO

A **Figura 6.12** apresenta os resultados de aceleração obtidos com as diferentes variações do **DTM** na mesma arquitetura, porém usando um mecanismo de despacho capaz de reusar em um mesmo ciclo cadeias de instruções e *traces* dependêntes.

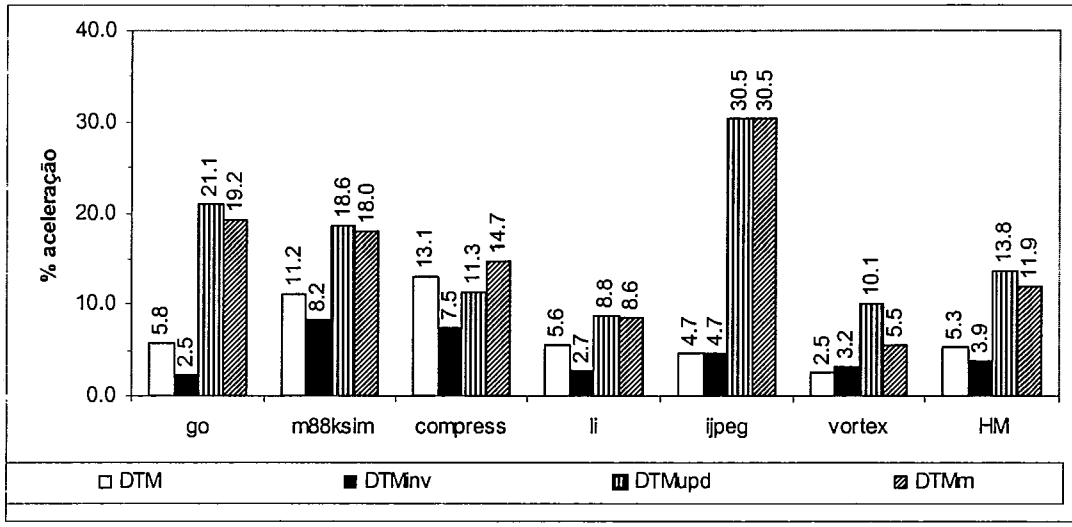


Figura 6.12: Aceleração com implementação sem verificação de dependência.

Podemos observar que os resultados obtidos com o **DTM*upd*** e **DTM*m*** fornecem um acréscimo de 7,2% e 6,5% respectivamente, sobre a média harmônica dos resultados anteriores. Existem dois fatores que justificam os ganhos de aceleração obtidos com esta implementação. Primeiro, o número excessivo de *traces* fragmentados por instruções ocupando *delay slot* pode ter produzido uma grande quantidade de cadeias de *traces* dependentes com poucas instruções e a adição de técnicas de reuso de cadeias de instruções e *traces* dependentes podem contribuir para compensar os problemas provocados por esta característica do conjunto de instruções do **Sparc v7**.

O segundo fator está relacionado com *traces* e instruções redundantes que possuem um subconjunto de registradores do contexto de entrada dependêntes do contexto de saída de um *trace* ou instrução redundante despachado no mesmo ciclo. Estes *traces* e instruções redundantes não serão reusados pois seus operandos de entrada

não estão prontos. Porém, com a identificação de cadeias de instruções e *traces* dependentes podemos reusar estes *traces* e instruções em um mesmo ciclo.

Desta forma verificamos que a adição de mecanismos capazes de explorar o reuso de cadeias de *traces* e instruções dependentes em um mesmo ciclo através da antecipação dos resultados dos operandos de entrada de *traces* e instruções redundantes no momento do despacho pode contribuir de forma significativa com o desempenho dos mecanismos de reuso apresentados.

Neste capítulo apresentamos os resultados obtidos com os experimentos realizados com as diferentes variações do mecanismo do **DTM**. No próximo capítulo veremos a conclusão dos resultados apresentados e as direções futuras de trabalho a serem exploradas.

Capítulo 7

Conclusões e Trabalhos Futuros

Os resultados obtidos neste trabalho demonstram que a inclusão de um mecanismo de reuso de instruções de acesso à memória com antecipação de valores das instruções de leitura fornece um acréscimo de 4,1% sobre a média harmônica obtida com o **DTM** para um subconjunto de programas do *benchmark* **SPECint '95** composto pelos programas **go, m88ksim, compress, li, ijpeg** e **vortex.**

Além disso, a implementação do **DTM** em uma arquitetura diferente da utilizada nos experimentos realizados em [8] demonstra que este mecanismo fornece ganho de aceleração independente da arquitetura usada, e que a diferença de aceleração observada está relacionada com as limitações impostas pelo conjunto de instruções adotado.

Concluímos também que a adição de instruções de acesso à memória diretamente nas tabelas do **DTM** usando um mecanismo de invalidação seletiva dos valores das instruções de *loads*, consome recursos do contexto de entrada e saída do *trace*, reduzindo o tamanho médio deles em função do aumento do número de *traces* finalizados por *overflow* de contexto, e reduzindo a freqüência de reuso dos *traces* com o aumento do tamanho de suas entradas. Além disso, a falta de uma política de seleção e escalonamento de *traces* reduz a qualidade dos *traces* selecionados para reuso. Assim, o estudo de novas políticas de escalonamento de *traces* podem trazer melhores resultados, uma vez que *traces* com melhor qualidade podem ser selecionados para reuso.

Também concluímos que a inclusão dos valores das operações de *stores* não trazem melhorias significativas ao mecanismo e reduzem a qualidade dos *traces*

presentes na **MEMO_TABLE_T**, aumentando a incidência de *traces* com apenas duas instruções, uma instrução simples e uma instrução de escrita, e a vantagem obtida com o armazenamento dos valores que serão escritos é ocultada pela espera sofrida na fila de acesso à memória. Desta forma, existe uma grande chance de que a remoção do valor de escrita das operações de *stores* possa trazer menos impacto sobre o mecanismo.

Os resultados também demonstram que a implementação do reuso de instruções de acesso à memória com antecipação de valores de *loads*, diretamente no **DTM** fornecem ganhos significativos de desempenho. Porém, a adição deste recurso nas tabelas de reuso do **DTM** aumentam o custo de implementação deste mecanismo. Assim, um novo mecanismo denominado **DTM*m*** foi proposto com o objetivo de reduzir o custo de implementação através da separação do mecanismo de reuso de instruções de *loads* das tabelas de reuso do mecanismo original. Experimentos realizados com o **DTM*m*** comprovam que a separação das instruções de *loads* traz simplificações ao mecanismo, reduzindo o custo de implementação e mantendo o desempenho.

Os experimentos também demonstram que a implementação de um mecanismo capaz de antecipar os valores de instruções de acesso à memória fornece um acréscimo na aceleração final de 7,2% sobre a média harmônica obtida inicialmente. Assim, a adição de mecanismos de previsão de valores associados ao **DTM*m*** podem aumentar significativamente o desempenho deste mecanismo antecipando valores aos *traces* selecionados durante o despacho.

O surgimento de *traces* redundantes durante a construção de um *trace* finaliza o *trace* em construção e a inclusão de mecanismos capazes de concatenar *traces* de instruções em formação com *traces* redundantes mantendo os dois na **MEMO_TABLE_T** pode trazer um significativo ganho de desempenho ao mecanismo.

Também os estudos comparativos entre o mecanismo atual e a inclusão de múltiplas instruções de *loads* e *stores* em um mesmo *trace*, e o dimensionamento das tabelas de reuso após a adição da **MEMO_TABLE_L** podem trazer resultados

significativos ao **DTM*m***, e a inclusão de um mecanismo para verificação de dependência de dados entre as instruções contidas nos *traces* em formação pode ser usado para permitir a inclusão de instruções de *loads* no interior de *traces* que não possuem instruções dependentes do resultado da instrução de acesso à memória.

Observamos também que a presença de instruções ocupando *delay slot* introduz uma grande quantidade de *traces* fragmentados e o uso de uma tabela de memorização de *traces* composta por 512 entradas pode estar limitando significativamente o potencial dos mecanismos implementados na arquitetura **Sparc v7**. Assim, o estudo de novos parâmetros de *hardware* para configuração das tabelas de reuso podem contribuir significativamente com o desempenho destes mecanismos nesta arquitetura.

Apêndice A

O Ambiente de Simulação - SuperSIM

Para execução das análises foi desenvolvido um simulador de uma máquina superescalar com *pipeline* de seis estágios que implementa o conjunto de instruções do processador **Sparc v7** da **Sun Microsystems** [30].

Este simulador foi inteiramente desenvolvido usando técnicas de modelagem orientada a objetos. Esta abordagem fornece flexibilidade na definição dos componentes da arquitetura, permitindo que rapidamente sejam implementados novos mecanismos.

Neste capítulo apresentaremos alguns aspectos do desenvolvimento do simulador tais como a arquitetura base e o conjunto de instruções utilizado. Veremos também os principais recursos implementados e que permitem a rápida depuração do sistema.

A.1 DESCRIÇÃO DA ARQUITETURA BASE

A arquitetura base consiste de uma máquina com endereçamento de 32 *bits* e memória acessada por *byte* no formato *big endian* e *pipeline* de seis estágios separados pelas atividades de busca de instruções (**IF**), decodificação (**ID**), despacho (**DS**), execução (**EX**), *forwarding* (**C**) e finalização (**WB**).

Todos os acessos à memória são feitos por instruções de *load/store* e trabalham diretamente sobre os registradores, e todas as instruções são alinhadas em 32 *bits*.

Existem dois bancos de registradores um para os registradores de propósito geral e outro para os registradores especiais da arquitetura. Ambos os bancos de registradores operam com o conceito de registradores reais e futuros possibilitando execuções especulativas de instruções.

Os registradores de propósito geral trabalham com o conceito de janela de registradores, onde 520 registradores de ponto fixo de 32 *bits* são organizados de tal forma que a qualquer momento apenas 32 registradores estejam disponíveis. Existem duas operações que permitem o deslocamento da janela de registradores que são usadas com a finalidade de armazenar e recuperar o contexto de chamada de sub-rotinas [30].

A.1.1 TIPOS DE OPERAÇÕES

No modelo implementado existem seis categorias de instruções: instruções de acesso à memória, instruções aritméticas e lógicas, operações de transferência de controle, operações de leitura e escrita dos registradores especiais, instruções de ponto flutuante, e operações de coprocessador [30].

Operações de Acesso à Memória

Qualquer registrador de propósito geral ou ponto flutuante pode ser carregado ou armazenado diretamente, exceto o registrador N que possui o valor fixo zero. Todo o acesso à memória é endereçado por registrador-registrador ou registrador-deslocamento e suportam acessos a *bytes*, meia palavra (16 bits), palavra (32 bits) e palavra dupla (64 bits).

Instruções de acesso à memória são armazenadas no formato *big endian* onde o *byte* de mais alta ordem é armazenado no endereço menor conforme a convenção de endereçamento apresentada na **Figura A.1** a seguir.

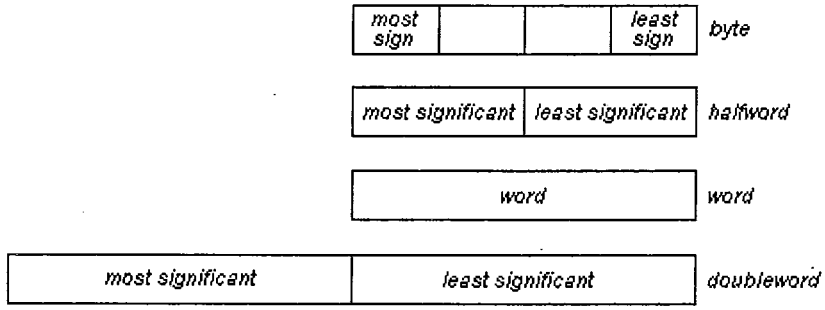


Figura A.1: Convenção de armazenamento de dados na memória.

As operações de acesso à memória são realizadas com alinhamento sobre o tipo de dado solicitado. Assim um acesso a um dado no formato *meia palavra*, *palavra* e *palavra dupla* devem ser realizados em endereços múltiplos de dois, quatro e oito respectivamente. O não cumprimento desta regra e o acesso fora do alinhamento correto provocam uma exceção. A **Tabela A.1** relaciona as instruções de acesso à memória presentes na arquitetura.

LDSB (LDSBA)	Carrega byte com sinal (usando outro espaço de memória)
LDSH (LDSHA)	Carrega meia palavra com sinal (usando outro espaço de memória)
LDUB (LDUBA)	Carrega byte sem sinal (usando outro espaço de memória)
LDUH (LDUHA)	Carrega meia palavra sem sinal (usando outro espaço de memória)
LD (LDA)	Carrega palavra (usando outro espaço de memória)
LDD (LDDA)	Carrega palavra dupla (usando outro espaço de memória)
LDF	Carrega valor de ponto flutuante
LDDF	Carrega valor de ponto flutuante com dupla precisão
LDFSR	Carrega registrador de estado das operações de ponto flutuante
LDC	Carrega palavra em registrador de coprocessamento
LDDC	Carrega palavra dupla em registrador de coprocessamento
LDCSR	Carrega registrador de estado das operações do coprocessador
STB (STBA)	Armazena byte (usando outro espaço de memória)
STH (STHA)	Armazena meia palavra (usando outro espaço de memória)
ST (STA)	Armazena palavra (usando outro espaço de memória)

STD (STDA)	Armazena palavra dupla (usando outro espaço de memória)
STF	Armazena valor de ponto flutuante
STDF	Armazena valor de ponto flutuante com dupla precisão
STFSR	Armazena registrador de estado das operações de ponto flutuante
STDFQ	Armazena conjunto de registradores de ponto flutuante
STC	Armazena palavra de registrador de coprocessamento
STDC	Armazena palavra dupla de registrador de coprocessamento
STCSR	Armazena registrador de estado das operações do coprocessamento
STDCQ	Armazena conjunto de registradores de coprocessor
LDSTUB (LDSTUBA)	Operação atômica em byte sem sinal (usando outro espaço de memória)
SWAP (SWAPA)	Operação de swap registrador-memoria (usando outro espaço de memória)

Tabela A.1: Instruções de acesso à memória.

Operações Aritméticas e Lógicas

Todas as instruções aritméticas e lógicas com inteiros podem ser realizadas entre registradores ou entre registrador e valor imediato de 13 *bits* com sinal. Durante uma operação o sinal do valor imediato é estendido em 32 *bits*.

Quando o registrador destino de uma operação for o registrador **R0** o resultado é descartado, porém o estado dos registradores de condição afetados são mantidos. A **Tabela A.2** apresenta o conjunto de instruções aritméticas e lógicas da arquitetura.

ADD (ADDcc)	Adição (com modificação de *icc*)
ADDX (ADDXcc)	Adição com *carry* (com modificação de *icc*)
TADDcc (TADDccTV)	Adição sinalizada com modificação de *icc* (*trap overflow*)
SUB (SUBcc)	Subtração (com modificação de *icc*)
SUBX (SUBXcc)	Subtração com *carry* (com modificação de *icc*)
TSUBcc (TSUBccTV)	Subtração sinalizada com modificação de *icc* (*trap overflow*)

MULScc	Passo de multiplicação com modificação de *icc*
AND (ANDcc)	'e' lógico (com modificação de *icc*)
ANDN (ANDNcc)	'não e' lógico (com modificação de *icc*)
OR (ORcc)	'ou' lógico (com modificação de *icc*)
ORN (ORNcc)	'não ou' lógico (com modificação de *icc*)
XOR (XORcc)	'ou exclusivo' lógico (com modificação de *icc*)
XNOR (XNORcc)	'não ou exclusivo' lógico (com modificação de *icc*)
SLL	Deslocamento lógico para a esquerda
SRL	Deslocamento lógico para a direita
SRA	Deslocamento aritmético para a direita
SETHI	Atribui valor de 22 *bits* aos *bits* mais significativos de um registrador
SAVE	Desloca a janela de registradores ativa armazenando o contexto
RESTORE	Desloca a janela de registradores ativa recuperando o contexto

Tabela A.2: Instruções aritméticas e lógicas da arquitetura.

Instruções de Transferência de Controle

Instruções de transferência de controle mudam o fluxo de execução de um programa. Existem cinco tipos básicos de instruções de transferência de controle: Desvios condicionais (**Bicc**, **FBfcc**, **CBccc**), desvios incondicionais (**JMPL**), chamadas de sub-rotinas (**CALL**), chamadas ao sistema operacional (**TRAP**) e retorno de chamadas ao sistema (**RETT**).

As instruções de transferências de controle podem ser organizadas em dois grupos: Desvio relativo ao valor do *program counter* (**PC**) ou indireto com base no valor de um registrador, ou ainda podem sofrer atraso ou não sofrer atraso (*delay slot*). A **Tabela A.3** a seguir apresenta a distribuição dos cinco tipos de desvios dentro desses grupos.

Bicc, FBfcc, CBccc	relativo ao PC	com atraso no desvio
CALL	relativo ao PC	com atraso no desvio
JMPL	baseado em registrador	com atraso no desvio
RETT	baseado em registrador	com atraso no desvio
Ticc	baseado em registrador	sem atraso no desvio

Tabela A.3: Distribuição dos grupos de desvios.

Desvios relativos ao **PC** são obtidos adicionando o endereço da instrução de desvio ao valor imediato com sinal multiplicado por quatro, para manter o endereço alvo alinhado com a palavra de 32 *bits*. Desvios condicionais são exemplos de desvios relativos ao **PC**, que testam os valores dos sinalizadores de condição e decidem por efetuar ou não o desvio.

Desvios indiretos baseados em registradores são computados pela soma dos valores dos registradores fonte ou pela soma do valor do registrador com um valor imediato com sinal.

Nas arquiteturas tradicionais as instruções de transferência de controle executam a instrução alvo imediatamente após a execução do desvio. Na arquitetura **Sparc v7** as instruções de desvio sofrem retardo e desta forma a instrução seguinte é executada antes da instrução alvo [30].

Esta instrução de transferência de controle recebe o nome de desvio com atraso e a instrução executada após o desvio é denominada *delay instruction.*

A **Listagem A.1** apresenta um exemplo de execução de desvio com retardo. Nesta listagem podemos ver que inicialmente é executada a instrução de adição **108** que está após a instrução de desvio **104** e somente após a instrução **108** é que a instrução **124** é executada.

```
100.  SUBCC R0, R1, #8
104.  BEQ $124
108.  ADD R1, R1, #1
            :
124.  OR R2, R1, R0
```

Listagem A.1: Exemplo de execução de desvio com atraso.

A arquitetura do **Sparc v7** também possui a capacidade de alterar o comportamento das instruções de desvio com atraso permitindo maior controle sobre este recurso. Este controle é realizado por um campo de sinalização localizado na instrução e denominado *annul bit*. Sempre que este campo estiver habilitado a *delay instruction* não será executada para um desvio tomado. A **Listagem A.2** apresenta um exemplo de desvio com *annul bit* habilitado.

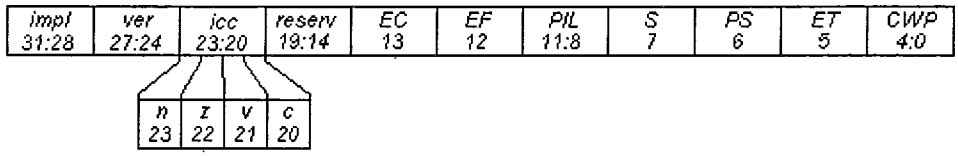
```
100.  SUBCC R0, R1, #8
104.  BEQ,a $124
;; delay instruction não é executada
            :
124.  OR R2, R1, R0
```

Listagem A.2: Exemplo de desvio com *annul bit*.

Operações Sobre Registradores Especiais

As operações sobre registradores especiais lêem e escrevem diretamente nos registradores **PSR**, **WIN**, **TBR**, **Y**, **FSR** e **CSR**, que descreveremos a seguir.

O registrador de estado do processador, **PSR,** é um registrador de 32 *bits* que armazena os valores dos sinalizadores de condição, os sinalizadores de presença de coprocessador e processador de ponto flutuante, além de sinalizar o modo supervisor e a janela de registradores corrente. A **Figura A.2** apresenta o formato dos campos deste registrador.



impl identificador do número da implementação do processador
ver identificador da versão da implementação
icc registradores de condicao
n sinalizador negativo
z sinalizador zero
v sinalizador overflow
c sinalizador carry
reserv campo reservado para futuras implementações
EC, EF sinalizador de presença de coprocessador e processador de ponto flutuante
PIL process interrupt level
S sinalizador de modo supervisor
PS, ET controle de excessões
CWP ponteiro para a janela de registradores corrente

Figura A.2: Registrador de estado do processador.

O controle dos limites da janela de registradores é realizado por uma mascara de 32 *bits* denominada *window invalid mask* **WIN**. Sempre que uma operação de **save** ou **restore** é executada, é verificado se o valor do *bit* correspondente à janela corrente está habilitado, se ele está habilitado uma exceção por *overflow* ou *underflow* da janela é gerada [30].

O registrador **TBR**, *trap base register*, fornece o endereço da função de tratamento associada com um determinado tipo de exceção.

Os registradores **FSR** e **CSR** são registradores de estado das operações de ponto flutuante e coprocessador respectivamente. Eles também armazenam os sinalizadores de condição usados por instruções de desvios.

A tabela a seguir, **Tabela A.4**, apresenta as instruções que manipulam os registradores especiais da máquina.

RDY, RDPSR, RDWIM, RDTBR	lê registrador Y (PSR, WIM e TBR)
WRY, WRPSR, WRWIN, WRTBR	escreve em Y (PSR, WIM e TBR)

Tabela A.4: Instruções que manipulam os registradores especiais.

Operações de Ponto Flutuante

As instruções de ponto flutuante efetuam computação com base em dois registradores fontes e atribuindo o resultado em outro registrador, a exceção fica por conta da instrução de comparação que atribui o resultado aos sinalizadores de condição *fcc*.

As operações podem ser realizadas sobre operandos de precisão simples ou dupla. Quando é requerida dupla precisão os registradores são manipulados aos pares, sendo o primeiro registrador um registrador de índice par (**F0, F2, ...**). A lista de instruções de ponto flutuante pode ser observada na **Tabela A.5**.

FiTOs, FiTOd, FiTOx	Converte valor inteiro em precisão simples (dupla ou estendida)
FsTOi, FdTOi, FxTOi	Converte precisão simples (dupla ou estendida) em valor inteiro
FMOVs	Move valores entre registradores de precisão simples
FNEGs	Obtém o valor simétrico do registrador de precisão simples
FABSs	Obtém o valor absoluto do registrador de precisão simples
FSQRTs, FSQRTd, FSQRTx	Raiz quadrada do valor de precisão simples (dupla ou estendida)
FADDs, FADDd, FADDx	Adição de valores de precisão simples (dupla ou estendida)
FSUBs, FSUBd, FSUBx	Subtração de valores de precisão simples (dupla ou estendida)
FMULs, FMULd, FMULx	Multiplicação de precisão simples (dupla ou estendida)
FDIVs, FDIVd, FDIVx	Divisão de precisão simples (dupla ou estendida)

FCMPs, FCMPd, FCMPx	Comparação de precisão simples (dupla ou estendida)
FCMPEs, FCMPEd, FCMPEx	Comparação de precisão simples (dupla ou estendida) com exceção.

Tabela A.5: Instruções de ponto flutuante da arquitetura.

A.1.2 FORMATO DAS INSTRUÇÕES

Todas as instruções são de 32 *bits* com os 2 *bits* mais significativos identificando o formato da instrução. Existem três formatos de instruções o **formato 1** apresentado na **Figura A.3**, é utilizado por instruções de chamadas a sub-rotinas, o **formato 2** é usados por instruções de desvio condicional e pela instrução **sethi** que atribui um valor constante a um registrador, e o **formato 3** usado pelo restante das operações [30].

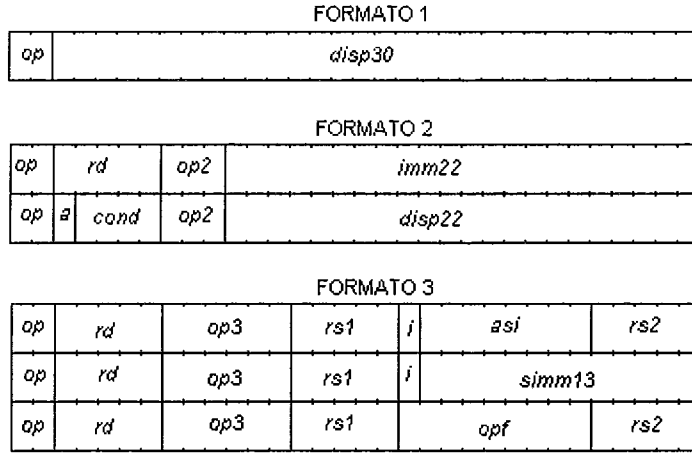


Figura A.3: Formato das instruções.

Na **Figura A.3** o campo **op** define um dos três formatos possíveis para a instrução e pode receber um dos valores apresentados na **Tabela A.6** abaixo.

Formato	Valor de (op)	Instruções
1	1	Call
2	0	Bicc, FBfcc, CBccc, SETHI
3	2 ou 3	Outras

Tabela A.6: Formato da instrução em função do valor de **op**.

A partir da definição do formato da instrução, se ela é do **formato 2** o campo **op2** será usado para determinar uma entre as cinco possíveis instruções que usam este formato. A **Tabela A.7** a seguir apresenta os valores que **op2** pode assumir.

Valor de (op2)	Instrução
0	UNIMP
2	Bicc
4	SETHI
6	FBfcc
7	CBccc

Tabela A.7: Instruções determinadas pelo valor de (**op2**).

Se a instrução for do **formato 3** ela é uma instrução lógica ou aritmética com dois operandos fontes e um destino ou com um operando fonte e um valor imediato de 13 *bits* com sinal, ou é uma instrução de ponto flutuante ou de coprocessador.

A.1.3 JANELA DE REGISTRADORES

A arquitetura **Sparc v7** usa o modelo de janela de registradores, onde um banco de registradores com 520 registradores de 32 *bits* são organizados de tal forma que os primeiros oito são globais e não mudam com a movimentação da janela, e os outros 512 registradores são divididos em 32 janelas sobrepostas de 24 registradores [30].

Usando janelas sobrepostas este mecanismo permite que o contexto de uma sub-rotina seja armazenado e recuperado com o deslocamento da janela como mostra a **Figura A.4**. Este mecanismo também permite o uso dos registradores em sobreposição para passagem de parâmetros entre funções.

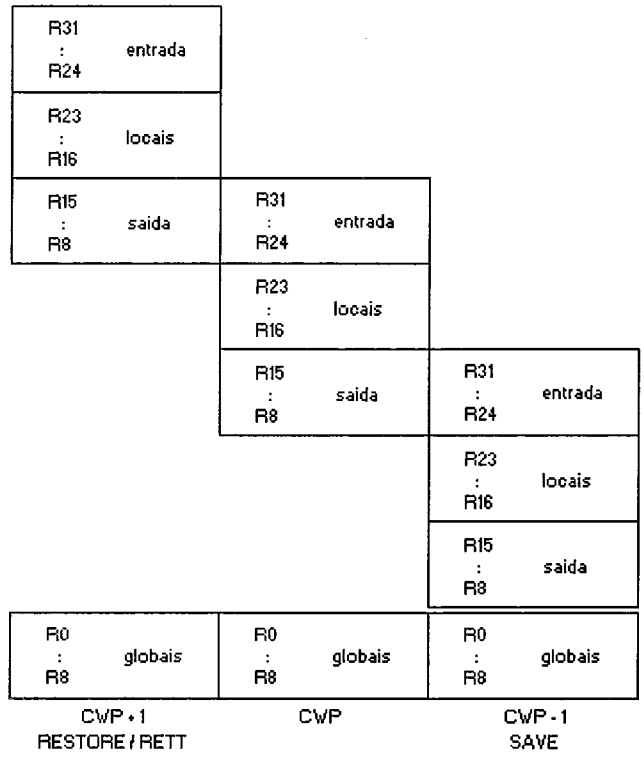


Figura A.4: Ilustração do mecanismo de janela de registradores.

Nas primeiras seções deste capítulo foi apresentada a arquitetura usada como base para o desenvolvimento do simulador. Nas próximas seções veremos a arquitetura implementada e alguns detalhes de implementação.

A.2 A ARQUITETURA DO SIMULADOR SUPERSIM

O simulador foi escrito inteiramente usando a linguagem **C++** e compilado com o compilador **GCC** para as plataformas **SUN Solaris 7** e **INTEL Linux**. No desenvolvimento do simulador foi usado como referência uma arquitetura superescalar

típica com *pipeline* de seis estágios, busca de instruções (*fetch*), decodificação (*decode*), despacho (*dispatch*), execução *(execute*), *forwarding* (*complete*) e finalização (*write-back*), conforme apresentado na **Figura A.5.**

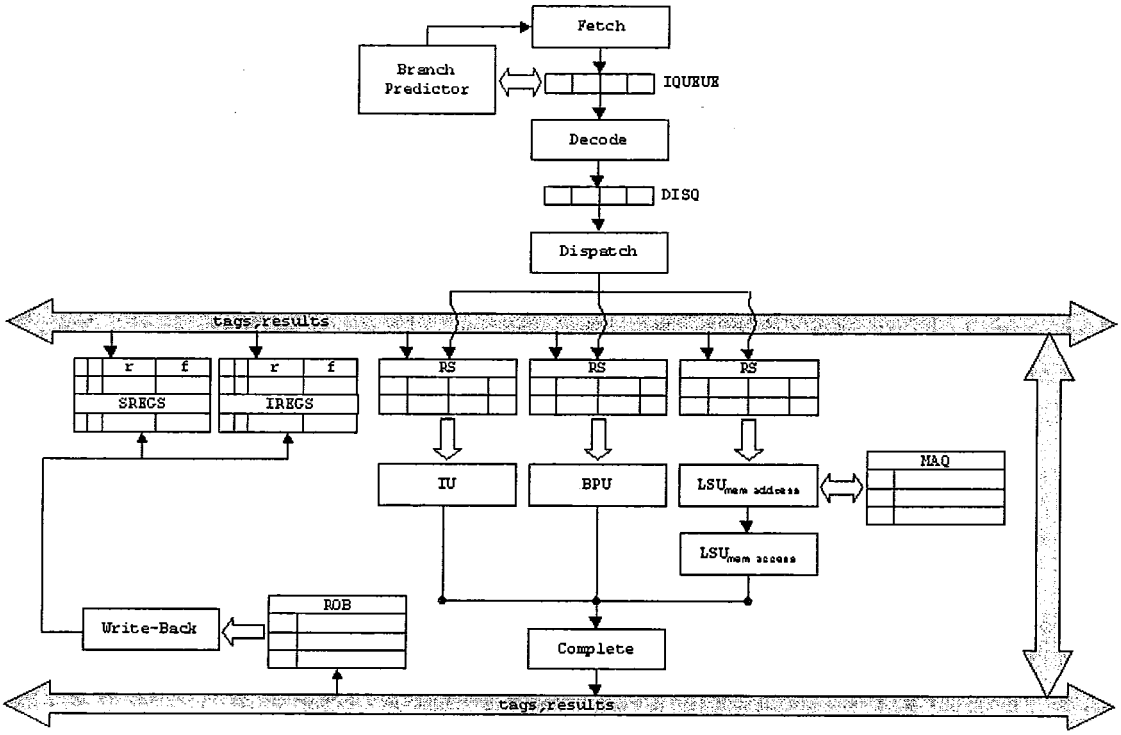


Figura A.5: *Pipeline* superescalar implementado pelo simulador SuperSIM.

A cada ciclo o estágio de busca acessa até quatro instruções consecutivas, efetua uma pré-decodificação e consulta o preditor de desvios para redirecionamento do ponteiro de instruções, **PC**. Instruções de desvios tomados finalizam o ciclo de busca após a obtenção da instrução de atraso (*delay instruction*).

No estágio de decodificação, as instruções são removidas da fila de instruções e decodificadas para em seguida serem inseridas na fila de despacho.

No estágio de despacho, as instruções são removidas da fila de despacho e os valores dos registradores fontes requeridos são acessados. Se os registradores fontes estiverem em uso um *tag* é retornado identificando a instrução que atualmente detém este

recurso. Em seguida os registradores destino da instrução são marcados como ocupado e um identificador da instrução é atribuído a eles. Depois as instruções são inseridas nas estações de reserva correspondentes e no *buffer* de reordenação. Instruções de acesso à memória são colocadas na fila de acesso à memória para manter a coerência do acesso de leitura e escrita.

O estágio de execução é compartilhado por três *pipelines* dedicados aos processamentos de instruções de inteiro, transferência de controle e acesso à memória.

Instruções aritméticas e lógicas seguem pelo *pipeline* de processamento de inteiros. Este *pipeline* possui apenas um estágio usado para a computação da operação e após serem executadas as instruções são inseridas em um registrador do *pipeline* para processamento pelo estágio seguinte, *forwarding*.

As operações de acesso à memória usam dois estágios independentes para execução da instrução. O primeiro estágio, denominado estágio de cálculo do endereçamento, acessa as instruções presentes na fila de acesso à memória em ordem e calcula os endereços das instruções que possuem operandos prontos.

O segundo estágio é responsável pelo acesso à memória e remove as instruções com endereço calculado em ordem da fila de acesso à memória conforme os seguintes critérios: Operações de leitura são efetuadas se não houver nenhuma instrução de escrita pendente, e as operações de escrita só são executadas se não existem instruções de escrita pendente e nenhuma instrução de transferência de controle que antecede ela e que ainda não foi executada.

Para simplificação da arquitetura todos os acessos à memória são resolvidos em um único ciclo como se a taxa de acerto na *cache* fosse de 100%. Está abordagem reduz os ganhos obtidos com os mecanismos implementados que atuam reusando operações de memória.

As instruções de transferência de controle são resolvidas no momento do despacho se os operandos de entrada estão disponíveis, caso os operandos não estejam disponíveis as instruções são encaminhadas às estações de reserva onde aguardam pelos valores dos operandos. À medida que os operandos ficam prontos às operações de transferência de controle vão sendo executadas.

Após a execução de uma operação de transferência de controle o resultado é comparado com a previsão inicial e se estiver correto o processamento continua normalmente. Porém se a previsão foi incorreta é realizado um *flush* nas filas de instrução, decodificação e despacho, as entradas posteriores ao desvio nas estações de reserva, *buffer* de reordenação e fila de acesso à memória são eliminadas, e os estados dos registradores e do ponteiro da janela de registradores recuperados.

Instruções de chamadas ao sistema operacional não são executadas no estágio de execução, mas sim no estágio de finalização. Está medida procura simplificar o tratamento de exceções executando estas em um momento em que o processador se encontra em um estado não especulativo.

O estágio de *forwarding* recebe as instruções executadas e atualiza os registradores futuros e as estações de reservas que aguardam estes valores. Em seguida os resultados destas instruções são atualizados no *buffer* de reordenação.

No estágio de finalização as instruções são retiradas em ordem e atualizam os registradores reais da arquitetura. Se a instrução retirada for uma instrução de chamada ao sistema operacional é efetuado um *flush* em toda a arquitetura e a chamada correspondente é executada como uma instrução atômica.

Nesta arquitetura as filas de instruções e despacho, as estações de reserva, a fila de acesso à memória e o *buffer* de reordenação possuem dezesseis entradas cada, com a finalidade de evitar que fatores externos limitem os mecanismos implementados.

A.2.1 MODELO DE DADOS DO SIMULADOR

Este simulador foi inteiramente implementado usando técnicas de modelagem orientada a objetos com o intuito de fornecer uma aplicação base que possa ser modificada e estendida com facilidade para suportar novos modelos de mecanismos. A **Figura A.6** apresenta a hierarquia de dados do simulador.

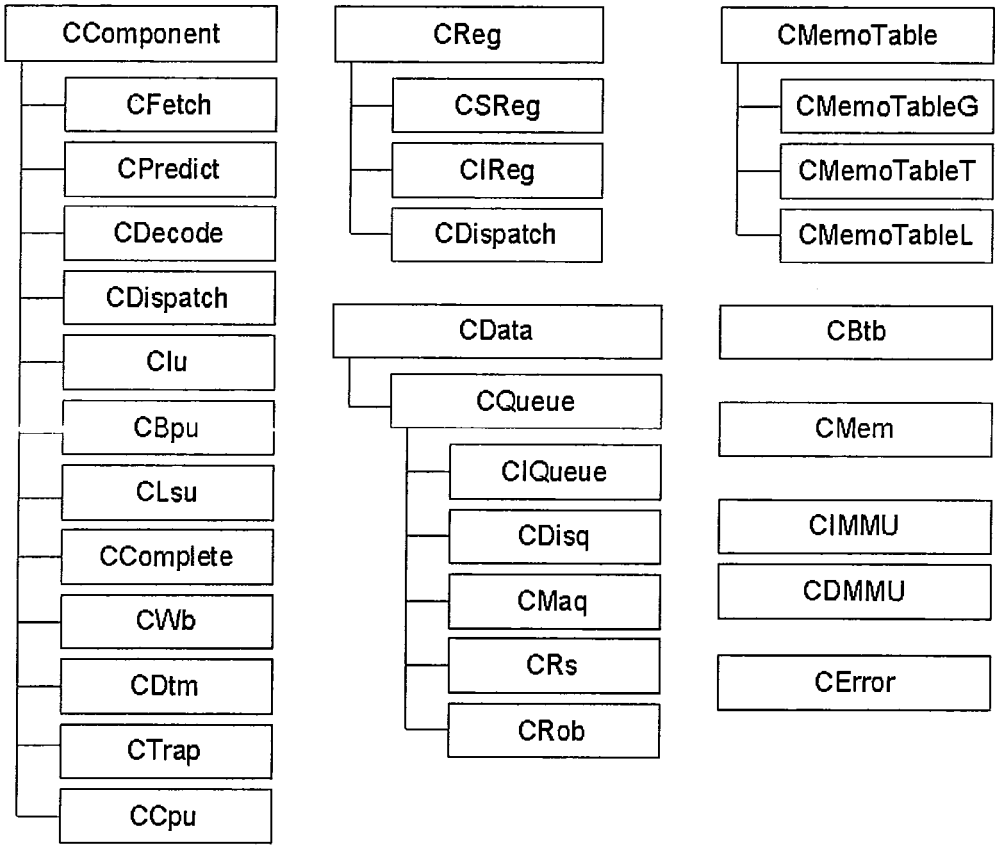


Figura A.6: Hierarquia de dados do simulador **SuperSIM**.

Conforme a **Figura A.6** o simulador é composto por três hierarquias básicas generalizadas pelas classes **CData** e **CReg**, que definem as estruturas de dados básicas do

simulador tais como o banco de registradores e filas de instruções, e pela classe **CComponent** que define as estruturas de controles, tais como as unidades de execução e o próprio processador.

A.2.2 RECURSOS IMPLEMENTADOS

Os recursos implementados foram baseados no estudo de várias arquiteturas existentes e o simulador foi desenvolvido procurando ampliar ao máximo a quantidade de recursos de modo a permitir a implementação dos mecanismos em análise sem necessidade de modificações.

Todos os parâmetros definidos no simulador como, por exemplo, o número de entradas na fila de instruções, a largura de despacho e o número de unidades funcionais são parametrizados durante a compilação.

Busca Antecipada de Instruções

O mecanismo de busca antecipada de instruções lê até quatro instruções por ciclo e insere em um *buffer*. Este mecanismo é bloqueado quando o *buffer* de instruções está cheio. Durante a fase de busca das instruções ocorre uma pré-decodificação que identificam desvios incondicionais e efetua a transferência para o novo endereço de busca das instruções.

Previsão de Desvios

A previsão de desvios é realizada através da técnica do *branch target buffer* (**BTB**) utilizando uma tabela associativa por conjunto com 16 *slots* de 8 entradas e dois *bits* de previsão. O algoritmo implementado pela **BTB** é uma adaptação do contador saturado. Quando a instrução de desvio não possui uma entrada na tabela é realizada a previsão estática assumindo o desvio como não-tomado.

Execução Especulativa

O número máximo de instruções de desvios que podem ser executadas especulativamente é limitado apenas pelo tamanho do *buffer* de reordenação. A integridade dos dados quanto à previsão incorreta de desvios é garantida por este mecanismo e pelos registradores futuros, sendo o controle realizado na fila de despacho com a adição de *bits* identificadores do nível de profundidade da execução especulativa.

Registradores Futuros

Cada registrador de propósito geral, **IREGS**, e de estado, **SREGS**, possuem um registrador futuro associado que garante as unidades de reserva que aguardam dados à disponibilidade dos mesmos com pelo menos um ciclo de antecedência, além de tornarem possível à execução especulativa de desvios.

Os registradores de condição por estarem contidos dentro da estrutura do registrador de estado **PSR** também estão associados a um registrador futuro, o que permite a resolução de desvios com antecedência.

Estações de Reserva

Estações de reserva armazenam as instruções que estão à espera de recursos. O simulador foi implementado de modo que o número de estações de reserva pudessem ser parametrizadas durante a compilação.

***Buffer* de Reordenação**

O *buffer* de reordenação garante a atualização correta dos registradores de propósito geral, **IREGS**, e dos registradores de estados, **SREGS**.

A.3 APRESENTAÇÃO DO SIMULADOR SUPERSIM

O simulador **SuperSIM** foi desenvolvido de modo a permitir uma visão de todo o processador em execução. O processamento dos programas pode ser passo-a-passo, para um determinado número de instruções, ou do ponto atual ao fim do programa ou até um ponto de parada determinado (*breakpoint*). Isto permite uma análise detalhada dos resultados. No modo passo-a-passo, por exemplo, podemos observar o fluxo de dados do simulador e acompanhar as mudanças nas estruturas de controle possibilitando a identificação de falhas nos algoritmos com facilidade.

Os programas usados no experimento podem ser carregados de dentro do ambiente e executados várias vezes, o que permite uma maior facilidade na análise de cada experimento.

Este simulador inclui recursos para criação de arquivo de *log* de instruções que apresenta um sumario das operações finalizadas, arquivo de *log* de blocos básicos que permite a comparação do fluxo de controle do programa e arquivo de *log* para intercambio de dados usado para validar grandes quantidades de instruções processadas com o resultado da execução fornecido pelo simulador executando o código sem os mecanismos implementados. Este *log* é gerado de forma binária e compactada permitindo o registro de um grande número de instruções.

Foram criados vários relatórios que permitem uma visão total do simulador durante a execução dos programas. Os relatórios criados correspondem à: *Program*, *Arguments*, *Memory*, BTB, *Fetch*, *Instruction Queue*, *Dispatch Queue*, IREG, SREG, *Reservation Stations*, IU, LSU, BPU, *Memory Access Queue*, *Reorder Buffer*, *Complete* e *Statistics*.

Program - Este relatório fornece uma visão do programa carregado na memória e permite o estabelecimento de pontos de parada para exame do estado da arquitetura e depuração do simulador.

Arguments - Apresenta a lista de argumentos passados ao programa que está em execução pelo simulador.

Memory - Este relatório reproduz o conteúdo da memória principal permitindo o monitoramento de instruções que modificam determinadas áreas da memória.

BTB - A inclusão de um relatório da BTB permite uma visão do funcionamento da tabela de destinos de desvios e acompanhamento das previsões permitindo um maior controle na depuração dos programas.

Fetch - A visualização do mecanismo de busca possibilita o acompanhamento das instruções que alimentam o *buffer* de instruções e a visão do resultado da pré-decodificação destas instruções.

Instruction Queue - A fila de instruções permite a observação do comportamento do mecanismo de busca e o monitoramento completo do *pipeline*.

Dispatch Queue - A visualização da fila de despacho permite analisar o mecanismo de despacho e a execução especulativa de desvios.

IREG - Através deste relatório é possível visualizar o banco de registradores de propósito geral, observar os registradores que estão ocupados à espera da conclusão de uma instrução e ver as modificações efetuadas nos registradores reais e futuros.

SREG - Permite a visualização dos registradores de controle e registradores futuros correspondentes e o acompanhamento na utilização destes recursos.

Reservation Stations - Este recurso permite observar as instruções nas estações de reserva, podendo acompanhar as instruções prontas para execução, as que estão à espera de operandos e as instruções em processamento.

IU, LSU e **BPU** - A janela de execução mostra uma visão das instruções que passaram pelo estágio de execução, mas ainda não foram concluídas. A observação desta janela e das janelas *Regs* e *GPR* permite a compreensão da execução do programa.

Memory Access Queue - Visualização da fila de acesso á memória permitindo o acompanhamento das instruções pendentes e da seqüência de instruções de acesso à memória executadas.

Reorder Buffer - Este relatório permite a visualização do *buffer* de reordenação onde observamos a ordem de atualização dos registradores e o fluxo de execução do programa. Também podemos observar as instruções que estão prontas para serem concluídas.

Complete - Apresenta a seqüência de instruções finalizadas pelo processador permitindo o acompanhamento da execução do programa e depuração do mesmo.

Statistics - Relatório instantâneo de informações estatísticas e medidas dos mecanismos do processador.

A.4 LIMITAÇÕES DO SIMULADOR

O simulador **SuperSIM** possui algumas limitações impostas pela falta de implementação de alguns recursos arquiteturais presentes na arquitetura base tais como, registradores de ponto-flutuante e coprocessamento, instruções de acesso em espaço de endereçamento diferente e exceções por *overflow* e *underflow* da janela de registradores.

Por não suporta operações de ponto-flutuante e instruções de *load* e *store* com registradores de ponto-flutuante e coprocessador, o simulador não suporta aplicações que fazem uso deste tipo de instruções e paralisa a execução do programa quando uma dessas instruções atinge o topo do *buffer* de reordenação.

A falta de um controle de exceção por *overflow* e *underflow* da janela de registradores inviabiliza o uso do mecanismo de janelas em aplicações altamente recursivas e na ocorrência deste evento o simulador paralisa a execução do programa. Porém podemos compilar as aplicações altamente recursivas desabilitando o uso intensivo do mecanismo de janelas de registradores.

Outra limitação do simulador com relação à arquitetura base se deve as cadeias de duplos desvios denominadas *delayed control-transfer couples* [30]. Estas cadeias de

duplos desvios possuem funcionalidades específicas para cada par de desvios encontrado e o simulador implementa apenas algumas dessas cadeias, paralisando a execução na ocorrência de uma cadeia que não pode ser resolvida.

Bibliografia

[1] BURGER, D., AUSTIN T. M., "The Simplescalar Tool Set, Version 2.0", *Technical Report 1342*, Computer Science Departament of University of Wisconsin.

[2] CALDER, B., GRUNWALD, D., "Next cache line and set prediction", In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp 287-296, Jun-1995.

[3] COSTA, A. T., FRANÇA, F. M. G., CHAVES, E. M. F., "The Reuse Potential of Trace Memoization", *Technical Report ES-498/99*, COPPE/UFRJ, Rio de Janeiro, May-1999.

[4] COSTA, A. T., FRANÇA, F. M. G., "Process of Formation Memorization and Reuse, in Execution Time, of Sequences of Dynamic Instructions in Computers", *International Patent, number WO 01/04746 A1, Patent Cooperation Treaty (PCT)*, Jul-1999.

[5] COSTA, A. T., FRANÇA, F. M. G., CHAVES, E. M. F., "Evaluating DTM in a Superscalar Processor Architecture", *Technical Report ES-538/00*, COPPE/UFRJ, Rio de Janeiro, Aug-2000.

[6] COSTA, A. T., FRANÇA, F. M. G., CHAVES, E. M. F., "The Dynamic Trace Memoization Reuse Technique", In *Proceedings of the International Conference on Parallel Architecture and Compiler Techniches* (PACT2000), Oct-2000.

[7] COSTA, A. T., FRANÇA, F. M. G., CHAVES, E. M. F., "Exploiting Reuse with Dynamic Trace Memoization: Evaluating Architectural Issues", In *Proceedings of the 12th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 163-172, Oct-2000.

[8] COSTA, A. T., "Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador", *D.Sc. dissertation*, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil, 2001.

[9] FRANKLIN, M., SOHI, G. S., "ARB: A Hardware Mechanism for Dynamic Reordering of Memory References", In *Proceedings of the IEEE Transaction Computers v.45*, n. 5, pp. 552-571, May-1996.

[10] GABBAY, F., MENDELSON, A., "Speculative Execution Based on Value Prediction", *Technical Report EE-TR 1080*, Technion – Israel Institute of Technology, Nov-1996.

[11] GABBAY, F., MENDELSON, A., "Can Program Profiling Support Value Prediction?", In *Proceedings of the 30 th Annual Symposium on Microarchitecture*, pp. 270-280, Dec-1997.

[12] GABBAY, F., MENDELSON, A., "The Effect of Instruction Fetch Bandwidth on Value Prediction", In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 272-281, 1998.

[13] GONZALEZ, A., TUBELLA, J., MOLINA, C., "Trace-Level Reuse", In *Proceedings of the International Conference on Parallel Processing*, pp 30-37, Japan, Sep-1999.

[14] HEIL, T. H., SMITH, Z., SMITH J. E., "Improving Branch Predictors by Correlating on Data Values", In *Proceedings of the 32 th International Symposium on Microarchitecture*, pp. 28-37, Nov-1999.

[15] HENNESSY, J., PATTERSON, D., *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmman, pp 29-38 e pp 76-83, 1997.

[16] HUANG, J., LILJA, D.J., "Exploiting Basic Block Value Locality with Block Reuse", In *Proceedings of the 5th High Performance Computer Architecture* (HPCA), pp 106-114, Jan-1999.

[17] HUANG, J., LILJA, D.J., "Exploiring Sub-Block Value Reuse for Superscalar Processors", In *Proceedings of the 2000 International Conference on Parallel Architecture and Compiler Techniches* (PACT2000), Oct-2000.

[18] JACOBSEN, E., ROTENBERG, E., SMITH, J. E., "Assign Confidence to Conditional Branch Predictions", In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 142-152, Dec-1996.

[19] LAM, M. S., WILSON, R. P., "Limits of Control Flow on Parallelism", In *Proceedings of the 19th International Symposium on Computer Architecture ACM/IEEE*, pp. 46-57, Jul-1992.

[20] 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 Programming Languages and Operating Systems* (ASPLOS), pp 138-147, Oct-1996.

[21] LIPASTI, M. H., SHEN, J. P., "Exceeding the Dataflow Limit Via Value Prediction", In *Proceedings of the 29th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp 226-237, Dec-1996.

[22] MACFARLING, S., "Combining Branch Predictors", *Technical Report TN-36*, DEC Western Research Laboratory, Jun-1993.

[23] MARTIN, M. M., ROTH, A., FISCHER, C. N., "Exploiting Dead Value Information", In *Proceedings of the 30 th Annual Symposium on Microarchitecture*, pp. 128-135, Dec-1997.

[24] MOSHOVOS, A., SOHI, G., "Streamlining Inter-operation Memory Comunication via Data Dependence Prediction", In *Proceedings of the 30th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp 235-245, Dec-1997.

[25] PAN, S.-T., SO, K., RAHMEH, J. T., "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pp. 76-83, Oct-1992.

[26] REBELLO, V. E. F., "Neurocom Project", *Technical Report ProTem-II CC*, CNPQ, Brasil, May-1997.

[27] 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. 24-34, Dec-1996.

[28] RYCHLIK, B., FAISTL J. W., J., KRUG, B. P., KURLAND, A. Y., SUNG, J. J., VELEK, M. N., SHEN, J. P., "Efficient and Accurate Value Prediction Using Dynamic Classification", *Technical Report of Microarchitecture Research Team in Dept. of Electrical and Computer Engineering*, Carnegie Mellon University, 1998.

[29] RYCHLIK, B., FAISTL, J., KRUG, B., SHEN, J. P., "Efficacy and Performance Impact of Value Prediction", In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Oct-1998.

[30] SPARC International, "The SPARC Architecture Manual, Version 7", *Sun Microsystems Inc.* Oct-1987.

[31] SAZEIDES, Y., SMITH, J. E., "The Predictability of Data Values", In *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 248-258, Dec-1997.

[32] SMITH, J. E., SOHI, G. S., "The Microarchitecture of Superscalar Processors", In *Proceedings of the IEEE Transaction Computers v.83*, n. 12, pp. 1609-1624, Dec-1995.

[33] SODANI, A., SOHI, G. S., "Dynamic Instruction Reuse", In *Proceedings of the 24th International Symposium on Computer Architecture* (ISCA), pp 194-205, Jun-1997.

[34] TOMASULO, R., "An Efficient Algotithm for Exploiting Multiple Arithmetic Units", *IBM Journal of Research and Development v. 11, n. 1*, pp. 25-33, Jan-1967.

[35] 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 7th ACM International Conference on Supercomputing*, pp. 67-76, Jul-1993.

[36] YEH, T.-Y., PATT, Y. N., "Two-Level Adaptive Branch Prediction", In *Proceedings of the 24 th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 51-61, Nov-1991.

[37] YEH, T.-Y., PATT, Y. N., "A Comparison of Dynamic Branch Predictors that Use Two Levels of Branch History", In *Proceedings of the International Symposium on Computer Architecture*, pp. 257-267, May-1993.

[38] TYSON, G. S., AUSTIN, T. M., "Improving the Accuracy and Performance of Memory Communication Through Renaming", In *Proceedings of the 30 th Annual Symposium on Microarchitecture*, pp. 218-227, Dec-1997.

[39] YUNG, R., "Design Decisions Influencing the UltraSPARC's Instruction Fetch Architecture", In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 178-190, Dec-1996.

[40] YOUNG, C., GLOY, N. C., SMITH, M. D., "A Comparative Analysis of Schemes for Correlated Branch Prediction", In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 276-286, Dec-1995.