UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LUIZ SEQUEIRA LAURINO

Reuso Especulativo de Traços com Instruções de Acesso à Memória

Dissertação apresentada como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Dr. Philippe Olivier Alexandre Navaux
Orientador

Porto Alegre, agosto de 2007

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Laurino, Luiz Sequeira

Reuso Especulativo de Traços com Instruções de Acesso à Memória / Luiz Sequeira Laurino. – Porto Alegre: PPGC da UFRGS, 2007.

99 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2007. Orientador: Philippe Olivier Alexandre Navaux.

1. Arquiteturas de Processadores. 2. Reuso de Valores. 3. Previsão de Valores. I. Navaux, Philippe Olivier Alexandre. II. Título.

A meus pais, Luiz e Miriam.

AGRADECIMENTOS

Em primeiro lugar, gostaria de agradecer à minha família, que sempre esteve muito presente em minha vida e me deu todo apoio e base necessários para seguir em frente. A meu pai Luiz, minha mãe Miriam, minha irmã Marcia, meu irmão Sergio, meus avós Nercy e Zilda, minha dinda Marta e meu primo Jomar, um agradecimento especial.

Agradeço aos colegas e amigos do laboratório, Ennes, Hermann, Cacique, Righi, Lucas, Pezzi, Marcia, Clarissa, ... e todos os demais que faziam daquele laboratório um bom lugar para tomar um chimas, jogar conversa fora e trabalhar também ;-). Embora na reta final deste trabalho nosso convívio não tenha sido tão freqüente, gostaria de dizer que o período em que desenvolvi atividades diárias no laboratório foi muito importante para meu crescimento, tanto pessoal como profissional. Algumas discussões que tivemos acerca de trabalho e outras nem tão sérias assim foram muito valiosas. As pedaladas de final de semana também serão sempre lembradas. Enfim, muito obrigado pela acolhida e apoio.

Não posso deixar de mencionar os meus colegas de grupo de pesquisa. Pilla, Pizzol, Tatiana, e todos os demais, muito obrigado pela ajuda, discussões, oportunidade de aprendizado e troca de conhecimentos com cada um de vocês durante a realização deste trabalho. Em especial, gostaria de expressar meus agradecimentos ao Pilla, por sua ajuda, apoio, idéias, discussões e disponibilidade em comparecer a várias reuniões realizadas ao longo da construção deste trabalho.

Agradeço aos meus amigos conterrâneos (“gurizada de Rio Grande”) que, alguns em situação semelhante à minha – alunos de Mestrado, passaram pelas mesmas angústias, dúvidas e superações. Além disso, foram muito importantes nos momentos de descontração durante este tempo de dissertação. Valeu Daniel, Clayton, Gonças, Patrícia, Velloso, De Bem, Rogério, Maurano, Mariana, Raquel, Ralph e todos os demais!

Gostaria também de agradecer a minha equipe de trabalho na Dell, que entendeu a importância deste trabalho para mim e não mediu esforços em fazer o possível para que sua conclusão fosse alcançada, muitas vezes tendo que tirar férias/dias de folga em períodos que não seriam os melhores para o nosso projeto. Ao Juarez, Quijano, Elson, Michael, e demais, meu muito obrigado.

Agradeço ao meu orientador, Prof. Navaux, pela confiança em mim depositada, orien-

tação, paciência em muitos momentos :-), e amizade durante este tempo junto ao PPGC. Navaux, muito obrigado por sua ajuda e acolhida.

Finalizando, devo dizer que este trabalho foi possível graças a recursos, na forma de bolsa de estudo, do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

BHB	*Block History Buffer*
BR	*Block Reuse*
DTM	*Dynamic Trace Memoization*
ILP	*Instruction Level Parallelism*
IPC	*Instructions per Cycle*
LRU	*Least Recently Used*
LSQ	*Load/Store Queue*
LVP	*Last Value Prediction*
PC	*Program Counter*
RB	*Reuse Buffer*
ROB	*Re-Order Buffer*
RST	*Reuse through Speculation on Traces*
RSTm	*Reuse through Speculation on Traces with Memory*
RT	*Recovery Table*
RUU	*Register Update Unit*
VP	*Value Prediction*
VPT	*Value Prediction Table*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Mesmo com o crescente esforço para a detecção e tratamento de instruções redundantes, as dependências verdadeiras ainda causam um grande atraso na execução dos programas. Mecanismos que utilizam técnicas de reuso e previsão de valores têm sido constantemente estudados como alternativa para estes problemas. Dentro desse contexto destaca-se a arquitetura RST (*Reuse through Speculation on Traces*), aliando essas duas técnicas e atingindo um aumento significativo no desempenho de microprocessadores superescalares.

A arquitetura RST original, no entanto, não considera instruções de acesso à memória como candidatas ao reuso. Desse modo, esse trabalho introduz um novo mecanismo de reuso e previsão de valores chamado **RSTm (*Reuse through Speculation on Traces with Memory*)**, que estende as funcionalidades do mecanismo original, com a adição de instruções de acesso à memória ao domínio de reuso da arquitetura. Dentre as soluções analisadas, optou-se pela utilização de uma tabela dedicada (Memo_Table_L) para o armazenamento das instruções de carga/escrita. Esta solução garante boa economia de *hardware*, não limita o número de instruções de acesso à memória por traço e, também, armazena tanto o endereço como seu respectivo valor.

Os experimentos, realizados com *benchmarks* do SPEC2000 *integer* e *floating-point*, mostram um crescimento de $2,97\%$ (média harmônica) no desempenho do RSTm sobre o mecanismo original e de $17,42\%$ sobre a arquitetura base. O ganho é resultado de uma combinação de diversos fatores: traços maiores (em média, $7,75$ instruções por traço; o RST original apresenta $3,17$ em média), embora com taxa de reuso de aproximadamente $10,88\%$ (inferior ao RST, que apresenta taxa de $15,23\%$); entretanto, a latência das instruções presentes nos traços do RSTm é maior e compensa a taxa de reuso inferior.

Palavras-chave: Arquiteturas de Processadores, Reuso de Valores, Previsão de Valores.

Speculative Trace Reuse with Memory Access Instructions

ABSTRACT

Even with the growing efforts to detect and handle redundant instructions, the true dependencies are still one of the bottlenecks of the computations. Value reuse and value prediction techniques have been studied in order to become an alternative to these issues. Following this approach, RST (Reuse through Speculation on Traces) combines both reuse mechanisms and has achieved some good performance improvements for superscalar processors.

However, the original RST mechanism does not consider load/store instructions as reuse candidates. Because of this, our work presents a new value reuse and value prediction technique named **RSTm (Reuse through Speculation on Traces with Memory)**, that extends RST and adds memory-access instructions to the reuse domain of the architecture. Among all studied solutions, we chose the approach of using a dedicated table (Memo_Table_L) to take care of the load/store instructions. This solution guarantees low hardware overhead, does not limit the number of memory-access instructions that could be stored for each trace and stores both the address and its value.

From our experiments, performed with SPEC2000 integer and floating-point benchmarks, RSTm can achieve average performance improvements (harmonic means) of $2,97\%$ over the original RST and $17,42\%$ over the baseline architecture. These performance improvements are due to several reasons: bigger traces (in average, $7,75$ per trace; the original RST has $3,17$ in average), with a reuse rate of around $10,88\%$ (less than RST, that presents reuse rate of $15,23\%$) because the latency of the instructions in the RSTm traces is bigger and compensates the smaller reuse rate.

Keywords: Processor Architectures, Value Reuse, Value Prediction.

1 INTRODUÇÃO

A crescente demanda por desempenho faz com que projetos de sistemas computacionais estejam cada vez mais complexos e sofisticados. *Softwares* muito mais arrojados são desenvolvidos regularmente e exigem processadores cada vez mais rápidos. Para aplicações de propósito geral, onde é difícil especializar componentes de *hardware*, o desafio de projeto é ainda maior e não é uma tarefa trivial.

Diante disso, diversas arquiteturas foram criadas com o intuito de melhorar ainda mais o desempenho de microprocessadores do estado da arte. Dentro desse contexto, destacam-se as arquiteturas superescalares. A idéia, aparentemente simples, é inserir várias unidades funcionais dentro do estágio de execução do *pipeline* e habilitar o processamento de várias instruções em paralelo. Com esse grande potencial, as arquiteturas superescalares sobressaíram-se entre as demais e hoje dominam o mercado. Microprocessadores como o Intel Pentium 4, AMD Athlon e Sun UltraSparc III são exemplos práticos do uso dessa arquitetura.

O número de unidades funcionais no estágio de execução varia de acordo com cada processador, mas já é comum encontrar oito ou mais unidades funcionais disponíveis. Mesmo contando com grande quantidade de hardware, o número de instruções executadas a cada ciclo (*Instructions Per Cycle* ou IPC) é baixo. Tipicamente, processadores do estado da arte não atingem, em média, IPC de 2 (HENNESSY; PATTERSON, 2003).

As dependências de dados verdadeiras e de controle são as principais responsáveis pela perda de desempenho, principalmente pelo fato de retardarem o início de uma computação. Nesses casos o incremento de recursos disponíveis em uma arquitetura não atenua o problema, visto que a crescente complexidade do hardware pode prejudicar a freqüência de operação do processador, além de não ser capaz de eliminar as dependências em questão.

A maioria dos mecanismos desenvolvidos para aumentar o desempenho de processadores superescalares foca sua atenção na busca por paralelismo em nível de instrução (ILP - *Instruction Level Parallelism*, em inglês). Entretanto, tais mecanismos não levam em conta a redundância naturalmente encontrada em programas, que acontece pelo fato de que grande parte deles são genéricos e, portanto, apresentam uma série de laços e

sub-rotinas, o que faz com que uma parcela significativa de código seja re-executada.

Estudos indicam que programas são constituídos de uma grande quantidade de computação redundante e previsível (GABBAY; MENDELSON, 1996; SODANI; SOHI, 1997; SAZEIDES; SMITH, 1997; SODANI; SOHI, 1998; BODIK; GUPTA; SOFFA, 1999), onde, por exemplo, muitas instruções executam as mesmas operações sobre o mesmo conjunto de entradas uma série de vezes. Estes estudos mostram que, para uma série de *benchmarks*, mais de 75% das instruções executadas produzem o mesmo resultado que execuções anteriores (SODANI; SOHI, 1998a). Tal característica provocou o desenvolvimento de várias técnicas (LIPASTI, 1997; COSTA, 2001; VIANA, 2002; PILLA, 2004a) baseadas no conceito de localidade de valores (LIPASTI; WILKERSON; SHEN, 1996) para ganhar desempenho através da exploração da redundância encontrada na execução de programas.

Em uma dessas técnicas, chamada RST (*Reuse through Speculation on Traces*) (PILLA, 2004a), é proposto o uso de duas metodologias simultaneamente: reuso e previsão de valores. Tal proposta prevê o uso conjunto das duas abordagens, de maneira a não se ter grande acréscimo de hardware se comparado a outras técnicas que utilizam as duas abordagens de forma isolada. O desenvolvimento deste mecanismo foi baseado no *Dynamic Trace Memoization* (DTM) (COSTA, 2001) onde é realizado apenas o reuso de traços, não sendo considerada a execução especulativa.

A atual implementação do RST, no entanto, não considera instruções de acesso à memória como parte do domínio de reuso. Instruções que acessam a memória necessitam de um tratamento especial para serem reusadas, o que não estava previsto na arquitetura RST original. Desse modo, o objetivo principal deste trabalho é analisar o impacto que o reuso de instruções de acesso à memória causa na arquitetura RST.

Neste trabalho é proposto o RSTm (*Reuse through Speculation on Traces with Memory*), que estende as características de funcionamento da técnica de reuso especulativo de traços (RST) (PILLA, 2004a). No RSTm o suporte a instruções de acesso à memória faz parte do conjunto de instruções pertencentes ao domínio de reuso. Dessa forma, instruções que acessam a memória poderão fazer parte dos traços, os quais tornar-se-ão maiores e com instruções cujas latências são mais altas. De modo geral, ganhos de desempenho podem ser esperados, pois, tendo traços mais longos (tanto com relação ao número de instruções quanto com relação à latência), a quantidade de instruções que deixarão de ser executadas será maior, causando, assim, a liberação do *pipeline* para outras instruções.

Vários trabalhos tratam do reuso ou previsão de traços com instruções de acesso à memória (REINMAN; CALDER, 1998; ONDER; GUPTA, 2001; JIN; CHO, 2006). Entretanto, nenhum deles combina o uso especulativo de traços com o reuso dos mesmos. O RSTm considera que a arquitetura pode trabalhar tanto com traços reusados quanto com traços previstos, ambos com a presença de instruções de acesso à memória.

As principais contribuições deste trabalho foram no sentido de compreender, deter-

minar e propor uma solução para mecanismos de reuso especulativos que, quando combinados a instruções de carga/escrita na memória e especialmente voltados para arquiteturas superescalares, possam gerar bons resultados. Além disso, provou-se que algumas características na formação dos traços fazem muita diferença no desempenho geral do mecanismo e que há balanceamento entre a quantidade de recursos disponível e o reuso obtido.

O RSTm apresenta bons resultados em relação ao RST original, tendo tido um aumento de desempenho de $2,97\%$ (média harmônica). Além disso, os traços formados com o mecanismo proposto são maiores, tendo em média $7,75$ instruções por traço, ao passo que o mecanismo anterior apresenta $3,17$. Uma análise completa dos resultados é feita ao final deste trabalho.

Este trabalho está organizado da seguinte forma; no Capítulo 2, são apresentados conceitos relacionados a reuso e previsão de valores, uma breve introdução aos processadores superescalares, bem como trabalhos relacionados e o estado da arte. Após, no Capítulo 3, é feita uma revisão a respeito do RST, mostrando suas principais características. No Capítulo 4 é detalhada a contribuição deste trabalho, mostrando a metodologia utilizada bem como o funcionamento do mecanismo para incluir instruções de acesso à memória no domínio de reuso da arquitetura RST. No Capítulo 5 o ambiente de simulação, os resultados alcançados e os limites do mecanismo são apresentados e discutidos. Por fim, são apresentadas algumas considerações finais e trabalhos futuros no Capítulo 6.

2 ESTADO DA ARTE E TRABALHOS CORRELATOS

Neste Capítulo são apresentados conceitos e trabalhos relacionados ao tema desta Dissertação. Na Seção 2.1 é feita uma revisão dos conceitos de arquiteturas superescalares. A seguir, na Seção 2.2, é abordado o conceito de localidade de valores. Em seguida, é detalhado o reuso de valores na Seção 2.3 e, ainda, a previsão de valores na Seção 2.4. Finalmente, na Seção 2.5, são revistos conceitos fundamentais a respeito do reuso e previsão de instruções de acesso à memória.

2.1 Arquiteturas Superescalares

Arquiteturas Superescalares exploram o paralelismo encontrado nos programas de computador e caracterizam-se por executar várias instruções escalares simultaneamente (HWANG, 1992; STALLINGS, 2002; ROSE; NAVAUX, 2003; HENNESSY; PATTERSON, 2003). Nestas arquiteturas, os processadores gerenciam múltiplos *pipelines* de instruções de modo que várias instruções possam ser executadas concorrentemente no mesmo ciclo de relógio. Tais *pipelines* alimentam uma série de unidades funcionais presentes nestes tipos de processadores, sendo que algumas, de acordo com sua área de ocupação e dissipação de potência, podem estar replicadas no processador.

Para que esta técnica possa ser implementada com sucesso, a unidade de busca e despacho de instruções deve ser capaz de buscar um número considerável de instruções a cada ciclo do relógio. Caso contrário, o *pipeline* ficará parado à espera de instruções a serem executadas, fazendo com que as unidades funcionais fiquem ociosas e subutilizando os recursos disponíveis do processador.

A Figura 2.1 mostra uma arquitetura superescalar conceitual (STALLINGS, 2002), onde as instruções de um programa são buscadas no estágio de busca. A seguir, elas são decodificadas e despachadas. Após analisadas as dependências existentes entre as instruções, aquelas que já têm suas entradas disponíveis são executadas, mesmo que em uma ordem distinta da encontrada no programa estático (desde que sejam respeitadas as dependências de dados verdadeiras). Por fim, o último estágio é responsável por retirar instruções na ordem original.

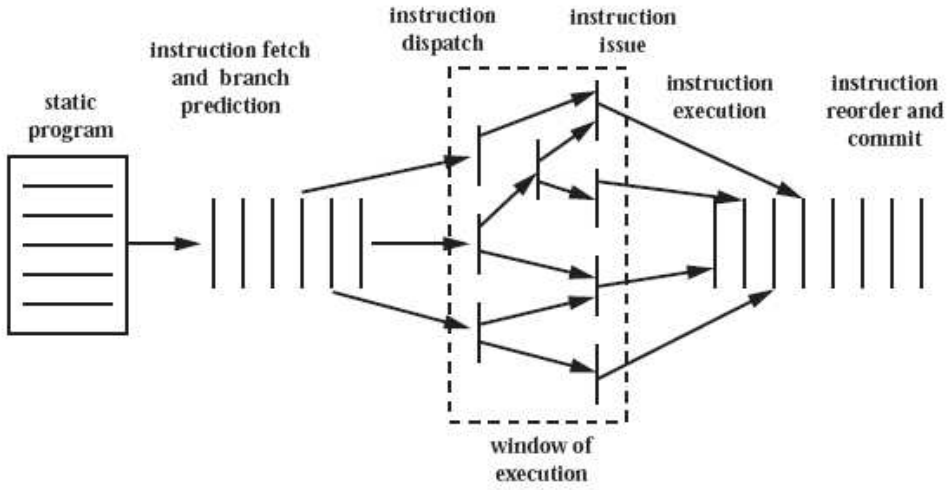

Figura 2.1: Arquitetura superescalar conceitual

Um conceito-chave nas arquiteturas superescalares é o paralelismo em nível de instrução ou ILP - *instruction level parallelism*, em inglês. O ILP diz respeito ao nível no qual as instruções de um programa podem ser executadas em paralelo. Para maximizar o ILP, muitas vezes é usada uma combinação de otimização através de compiladores e técnicas de *hardware*. ILP permite que o compilador e o *hardware* sobreponham a execução de múltiplas instruções ou até mesmo mudem a ordem em que as mesmas são executadas.

A quantidade de paralelismo em nível de instrução existente em um programa depende muito da aplicação. Em determinadas áreas, como computação gráfica e científica, a quantidade de ILP é bastante grande.

O paralelismo em nível de instrução apresenta algumas limitações (dependências), sendo que estas não podem ser totalmente resolvidas nem pelo compilador nem pelo *hardware*. Tais dependências surgem do fato de que os programas convencionais são escritos para serem executados seqüencialmente. Neste modelo, as instruções são executadas uma após a outra, de forma atômica (em um determinado instante, apenas uma instrução estará sendo executada) e na ordem especificada pelo programa.

Dito isto, podemos ter dois tipos de dependências: dados e controle. A seguir são listadas as dependências de dados (JOHNSON, 1991).

1. Dependência de dados ou RAW (*Read after Write*)
2. Dependência de saída ou WAW (*Write after Write*)
3. Anti-dependência ou WAR (*Write after Read*)

A dependência de dados, também chamada de dependência verdadeira, não pode ser resolvida nem por um compilador nem pelo *hardware*. Instruções que apresentem tal

dependência devem ter sua ordem de execução mantida. Já as outras duas dependências podem ser resolvidas pelo *hardware*, por exemplo. No caso da dependência de saída, a ordem de execução de instruções afeta o resultado de uma variável. Na anti-dependência, uma instrução utiliza uma variável que é posteriormente atualizada. Em ambos casos, o processador pode-se valer de uma técnica chamada renomeação de registradores (STALLINGS, 2002) para resolver tais dependências e aumentar, assim, o ILP.

Dependências de controle são aquelas em que uma instrução *A* depende de outra *B* se *B* determina quando *A* deve ou não ser executada. Este tipo de dependência normalmente está associada a desvios condicionais.

2.2 Localidade de Valores

O conceito de localidade de valores surgiu a partir dos trabalhos de Lipasti e Shen (LIPASTI; WILKERSON; SHEN, 1996; LIPASTI; SHEN, 1996; LIPASTI, 1997). Este conceito pode ser definido como a probabilidade que um valor já observado em uma sequência de instruções se repita em uma unidade de armazenamento ao longo da execução de um conjunto de instruções.

É interessante observar que operações envolvendo a memória apresentam, segundo os resultados de Lipasti, Wilkerson e Shen (1996), 50% de localidade de valor, em média, para um histórico do último resultado dessas operações. Para um histórico que armazena os últimos dezesseis resultados, o percentual de localidade de valor alcança, em média, 80%. Estes resultados são mostrados na Figura 2.2 (VIANA, 2002), onde são apresentados valores para a localidade de valores para instruções de *load* em vários *benchmarks*, no Alpha AXP e no PowerPC.

Os *benchmarks* apresentados são de inteiros do SPEC 92, SPECint 95 e de ponto flutuante do SPEC 92. É interessante observar que os *loads* de endereço apresentam maior localidade de valores do que os *loads* de dados.

Os mecanismos de previsão de desvios (YEH; PATT, 1991, 1992; MCFARLING, 1993; LEE; SMITH, 1984; HENNESSY; PATTERSON, 2003) também exploram a localidade de valores. Estes mecanismos têm como principal função determinar qual será o endereço a partir do qual a execução de um programa prosseguirá, após a tomada de um desvio. Para tanto, informações a respeito de instruções são obtidas e armazenadas. De acordo com tais informações, os desvios podem ser previstos como “tomados” ou “não-tomados”. Diversos trabalhos (JACOBSEN; ROTENBERG; SMITH, 1996; SODANI; SOHI, 1997; GABBAY; MENDELSON, 1998; HEIL; SMITH; SMITH, 1999; ONDER; GUPTA, 2001) corroboram a idéia de que localidade de valores é um comportamento extremamente presente nos programas e que podem ser explorados a fim de se obter ganhos de desempenho.

Sodani (2000) define que a **repetição dinâmica de instruções** se dá quando uma ins-

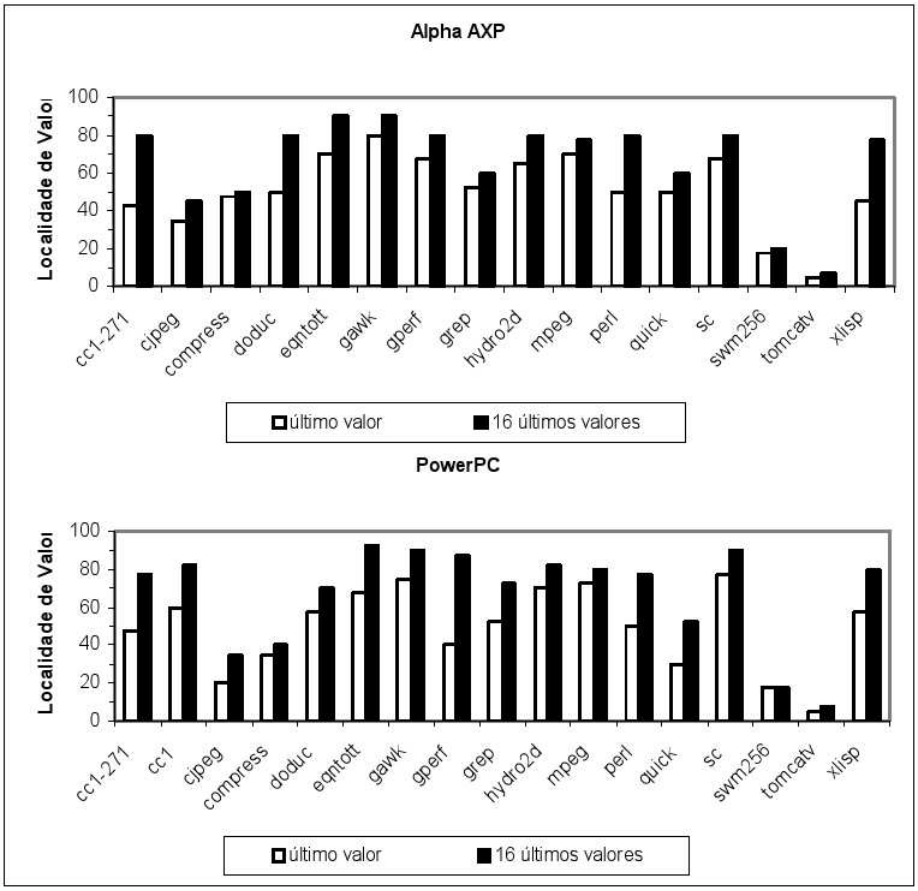

Figura 2.2: Localidade de valores para o Alpha APX e PowerPC

trução é executada diversas vezes com o mesmo conjunto de dados de entrada e, portanto, gera o mesmo conjunto de dados de saída como resposta. Esta definição está baseada no fenômeno que os programas têm de apresentar diversas formas de localidade de valores ao longo de sua execução, podendo ter dados sendo gerados repetidamente e instruções sendo executadas com os mesmos operandos. Seguindo esta linha, Pilla (2004a) amplia a definição e estabelece a **redundância de valores** como a repetição de resultados, gerados por instruções, blocos básicos ou traços de instruções quando o mesmo conjunto de entradas é submetido a execução.

A localidade de valores ocorre pelo fato de que programas genéricos freqüentemente apresentam *loops* e sub-rotinas que manipulam conjuntos de dados ao longo de sua execução. Além disso, dependendo da natureza do programa, os conjuntos de dados podem apresentar pouca variabilidade ao longo do tempo. A isso também são acrescidas as constantes, que normalmente não são detectáveis em tempo de compilação quando definidas em tempo de execução. Detalhes a respeito das diversas outras razões pelas quais os programas apresentam localidade de valores são explicados em (LIPASTI; WILKERSON; SHEN, 1996).

Neste contexto, vale salientar a existência de duas características geralmente observadas nos programas: localidade temporal e localidade espacial. O projeto de memórias

cache (número de blocos, associatividade, algoritmos de substituição, etc.) leva em conta as duas propriedades com o objetivo de diminuir o número de acessos à memória principal. Além disso, essas duas propriedades influenciam o reuso de valores:

- **Localidade temporal** é a propriedade que um dado endereço recentemente referenciado tem de ser acessado novamente em um curto intervalo de tempo.

- **Localidade espacial** é a propriedade que um dado endereço, próximo a outro recentemente referenciado, tem de ser acessado em um curto espaço de tempo. Do mesmo modo que na localidade temporal, a probabilidade de um dado contíguo a um valor recentemente acessado ser referenciado é grande para programas genéricos.

Mesmo com o uso de técnicas de compilação sofisticadas, onde diversas otimizações podem ser feitas no código, a localidade de valores continua presente, podendo em alguns casos até ser aumentada pelas modificações realizadas. Algumas referências que em tempo de compilação são desconhecidas podem se transformar em constantes durante a execução (LIPASTI, 1997). Além disso, o conteúdo de uma variável, por exemplo, pode forçar que um determinado cálculo seja repetido várias vezes com as mesmas entradas.

Diversos trabalhos têm explorado a localidade de valores e a redundância naturalmente encontrada em programas com o objetivo de minimizar a necessidade de execução repetida de instruções cujas entradas são as mesmas e, por conseqüência, proporcionar melhor desempenho do processador.

2.3 Reuso de Valores

O reuso de valores (LIPASTI; WILKERSON; SHEN, 1996; SODANI; SOHI, 1997; SAZEIDES; SMITH, 1997; SODANI; SOHI, 1998a; SODANI, 2000) é uma técnica não-especulativa que beneficia-se do conceito de localidade de valores e explora a redundância encontrada nos programas através da utilização dos resultados de execuções anteriores de instruções, e, portanto, não processa as instruções nas execuções subseqüentes. Dessa forma, a técnica implementa o reuso de um valor que já tenha sido previamente calculado. Tal idéia está baseada no fato de que as instruções freqüentemente são executadas com as mesmas entradas e, portanto, acabam gerando as mesmas saídas.

O funcionamento de um mecanismo de reuso de valores é basicamente o seguinte: assim que uma computação é executada pela primeira vez, suas entradas e saídas (resultados) são armazenados em uma tabela. Quando uma computação é encontrada novamente, as entradas são lidas da tabela e comparadas (teste de reuso) com aquelas em execução. O teste de reuso valida resultados quando o contexto de entrada (todos os operandos de entrada) previamente armazenado é o mesmo daquela computação que está em execução.

Em alguns casos as entradas são sempre as mesmas, o que poderia implicar numa otimização feita pelo compilador. Entretanto, estas otimizações acabam não sendo feitas pelo compilador por uma série de questões (SODANI, 2000): *(i)* o caminho dinâmico executado não é conhecido em tempo de compilação, pois pode depender de certas entradas; *(ii)* o compilador pode evitar a otimização do código a fim de manter sua corretude; *(iii)* a análise necessária para detectar repetições é complexa e difícil de ser realizada; *(iv)* algumas repetições ocorrem devido a limitações do conjunto de instruções e não podem ser eliminadas.

Segundo Sodani e Sohi (1998a), não é impossível ter-se reuso estático. Entretanto, os compiladores precisariam de propagação de constantes, eliminação de subexpressões comuns, linearização de laços, dentre outras técnicas. Todas essas técnicas seriam usadas de forma global, e um número bastante grande de registradores seria necessário.

O reuso de valores pode aumentar o desempenho de um programa das seguintes formas (SODANI, 2000):

- Instruções reusadas não são executadas;
- Computações úteis, mas em caminhos errados, podem ser preservadas;
- Os resultados ficam disponíveis mais rapidamente – possibilidade de computações dependentes começarem a ser executadas mais cedo;
- Dependências verdadeiras podem ser eliminadas, visto que, em alguns casos, computações dependentes podem executar em paralelo.

A principal desvantagem do reuso de valores é de que todas as entradas de uma computação devem estar disponíveis no momento do teste de reuso, onde a comparação das entradas deve ser feita com valores previamente armazenados. O reuso de uma computação ocorre apenas quando todas as entradas em execução são idênticas àquelas armazenadas. Por isso, algumas computações que seriam reusadas acabam não sendo, visto que alguma das entradas não está disponível para ser testada no momento apropriado.

Existem mecanismos especializados em reuso dinâmico de instruções (*dynamic instruction reuse*) (SODANI; SOHI, 1997, 1998,a), reuso de blocos básicos (*block reuse*) (HUANG; LILJA, 1999), reuso de traços de instruções (*trace reuse*) (GONZÁLEZ; TUBELLA; MOLINA, 1999; COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001), assim como blocos de instruções e sub-blocos de tamanhos variáveis (*sub-block reuse*). Analisaremos os principais a seguir.

2.3.1 Reuso Dinâmico de Instruções

Reuso Dinâmico de Instruções (SODANI; SOHI, 1998,a; SODANI, 2000) é uma técnica que explora o conceito de localidade de valores através do reuso de resultados já

disponíveis. A técnica utiliza um *Reuse Buffer* (RB) para armazenar as instruções redundantes. O RB nada mais é do que uma tabela utilizada para armazenar tanto as entradas quanto os resultados das instruções. Esta tabela ainda possui um mecanismo capaz de efetuar invalidações seletivas das entradas de acordo com a ocorrência de determinados eventos. A Figura 2.3 apresenta a estrutura genérica de um RB para reuso de instruções. As mesmas são armazenadas neste *buffer* e são acessadas através do PC (*Program Counter*) e, após teste de reuso, são determinadas quais instâncias podem ser reusadas. Instruções reusadas são enviadas diretamente para o estágio de confirmação de instruções, como pode ser visto na Figura 2.5.

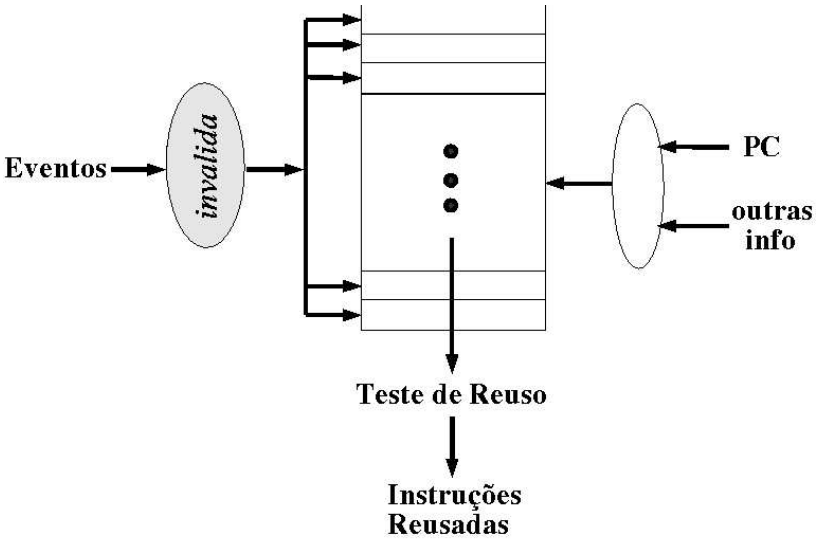

Figura 2.3: *Reuse Buffer* utilizado no reuso de instruções

A associatividade do RB determina quantas instâncias da mesma instrução podem ser armazenadas num dado momento. Um aumento na associatividade implica diretamente no crescimento do número de testes de reuso feitos. Esta análise é análoga àquela feita para memórias *cache*.

Sodani e Sohi (1998a; 2000) classificam o reuso dinâmico de intruções em quatro tipos. A Figura 2.4 mostra o formato das entradas no *Reuse Buffer* para cada um dos tipos, sendo que o formato dos dois últimos é o mesmo. Pode-se observar que cada entrada é composta por um campo **tag** de identificação (obtido a partir do PC), pelos campos **operand1** e **operand2**, que representam as entradas de cada uma das instruções. Além disso, o campo **res** guarda o resultado da instrução, **addr** contém o endereço de acesso à memória para instruções de leitura e alguns bits de sinalização indicam se o valor do campo **res** é ou não válido. Vejamos agora cada um dos quatro tipos classificados por Sodani e Sohi:

- S_v - é a implementação direta do reuso de instruções, onde os valores dos operandos de entrada são armazenados diretamente no RB e utilizados no teste de reuso;

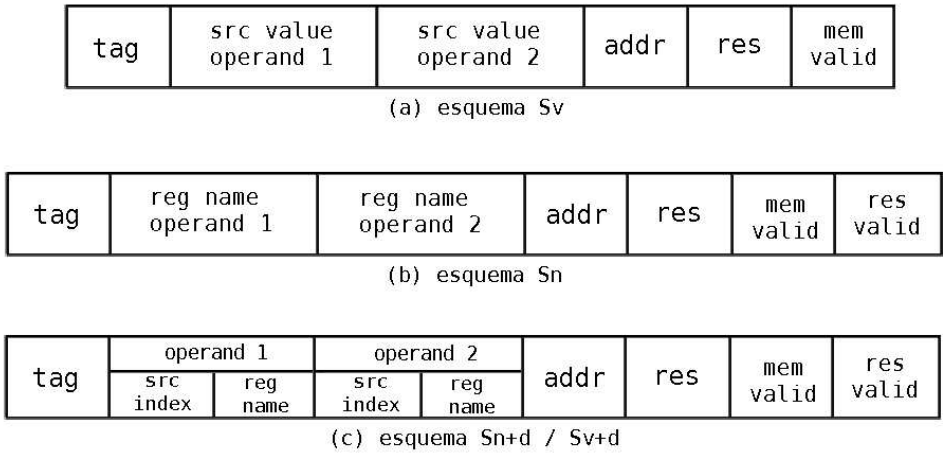

Figura 2.4: Entradas de um *Reuse Buffer*

- S_n - neste caso, os campos do RB armazenam os identificadores dos operandos fonte da instrução, e não seus respectivos valores. Dessa forma, o teste de reuso resume-se a validar que o valor dos registradores não se alterou desde que a RB foi atualizada para esta instrução;

- S_{v+d} - estende o primeiro esquema através da adição de verificações para determinar dependências entre instruções. A idéia é identificar a relação de dependência entre as instruções para evitar a invalidação demasiada das entradas do RB - que acontece frequentemente no esquema S_v. O mesmo propósito vale para S_{n+d};

- S_{n+d} - estende o segundo esquema, também, através da adição de verificações para determinar dependências entre instruções;

O funcionamento do reuso dinâmico de instruções pode ser assim descrito:

1. Assim que uma instrução é executada pela primeira vez, suas entradas e resultados são armazenados no RB;

2. À medida que a computação avança e uma instrução é encontrada novamente, as entradas são lidas do RB (em paralelo com a busca de instruções) e comparadas, através de um teste de reuso, com aquelas em execução (em paralelo com a decodificação da instrução);

3. O teste de reuso valida resultados quando o contexto de entrada (todos os operandos de entrada) previamente armazenado é o mesmo daquela instrução que está em execução. Aqui deve ser observado um dos esquemas (S_v, S_n, S_{v+d} e S_{n+d}) para realizar o teste de reuso;

4. Quando há um *match* entre os contextos de entrada, a instrução é dita reusada (não é executada) e é colocada na fila de finalização/confirmação de instruções (*commit*).

O reuso de instruções elimina dependências verdadeiras, visto que executa no mesmo ciclo várias instruções que normalmente executariam sequencialmente. A Figura 2.5 mostra basicamente o caminho que uma instrução percorre no *pipeline* caso seja executada ou reusada.

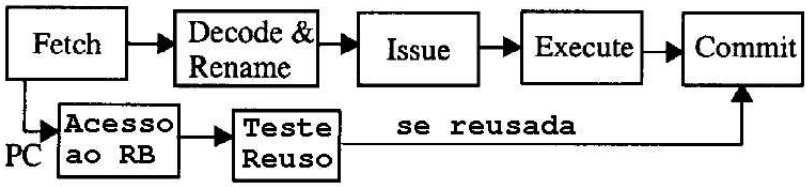

Figura 2.5: Instrução em *pipeline* com reuso de valores

2.3.2 Reuso de Blocos Básicos

Considerando um tamanho diferente para reuso, tem-se os blocos básicos - seqüências dinâmicas de instruções sendo que, se houverem desvios, eles serão a última instrução do bloco. Huang e Lilja (1999) apresentaram a idéia de se reusar um bloco básico, visto que suas entradas e saídas apresentariam grande localidade de valores. Tais blocos seriam identificados e teriam suas entradas e saídas armazenadas. Vale ressaltar que as entradas e saídas intermediárias ao bloco não precisam (e não são) guardadas. Blocos sem saídas indicam que não existe instrução que utilize-se de seus resultados, sendo, portanto, desnecessária sua execução.

O mecanismo de *block reuse* (BR), proposto em (HUANG; LILJA, 1999), possui uma tabela, denominada *Block History Buffer* (BHB), que é usada para armazenar os blocos básicos e suas entradas e saídas. A Figura 2.6 mostra como uma BHB pode estar organizada. Cada entrada é identificada por uma **tag**, obtida a partir do endereço da primeira instrução do bloco básico. Além disso, cada entrada possui espaço para armazenar os contextos de entrada e de saída (**Reg-In** e **Reg-Out**, respectivamente), bem como informações refentes à leitura e gravação na memória que seriam executadas pelas instruções do bloco básico (**Mem-In** e **Mem-Out**, respectivamente). Por último, um ponteiro (**Next-block**) identifica o endereço da instrução subseqüente ao bloco em questão.

Huang e Lilja (1999) mostraram ainda que, com uma BHB de 2048 entradas com 4 registradores de entrada, 5 registradores de saída, 4 entradas para memória e 2 saídas para memória, 90% de todos os blocos possíveis de serem reusados eram considerados para tal. Com essa configuração, os autores obtiveram ganhos da ordem de 9% no desempenho total das aplicações. Em seguida, Huang e Lilja (2000) propõem o reuso de *Sub-Block*, que são menores que os blocos básicos convencionais e podem ser construídos levando-se em conta o número de entradas e saídas. Entretanto, os *Sub-Blocks* ainda devem respeitar as restrições anteriores, onde um desvio deve, obrigatoriamente, ser a última instrução do bloco.

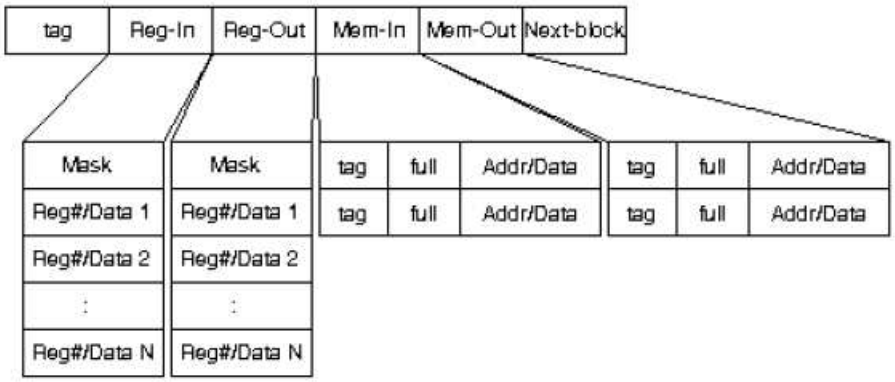

Figura 2.6: Exemplo de entrada para um *Block History Buffer* (BHB)

O mecanismo de reuso de blocos básicos inicial funciona basicamente da seguinte maneira:

1. Quando um endereço de início de bloco é obtido, a BHB é consultada a procura das instâncias deste bloco;

2. Assim que são encontradas, os valores dos registradores de entrada são comparados com os valores armazenados;

3. Se existirem valores válidos em **Mem-In**, a memória *cache* é consultada e os valores armazenados também são comparados com os da *cache*;

4. Se há um *match* nos valores de entrada e da *cache*, uma instância do bloco básico é reusada. Caso contrário, a instrução é despachada normalmente.

No caso do reuso ter ocorrido, as instruções pertencentes ao bloco são enviadas para confirmação (*commit*), os registradores de saída são atualizados e o ponteiro **Next-block** é usado para atualizar o PC. Se houver algum valor em **Mem-Out**, este é atualizado na *cache*.

Para minimizar o número de registradores de saída, algumas instruções têm seu formato modificado para evitar a ocorrência de *dead outputs*, ou saídas que não terão utilizadade. Neste caso, esta abordagem geralmente não apresenta compatibilidade com código legado (COSTA; FRANÇA; CHAVES FILHO, 2000a).

2.3.3 Reuso de Traços de Instruções

O reuso de traços de instruções (GONZÁLEZ; TUBELLA; MOLINA, 1999; COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001; PILLA, 2004a; PILLA et al., 2004; LAURINO et al., 2005; PILLA et al., 2006) é uma outra abordagem para o reuso de valores. Esta técnica é baseada no reuso de seqüências dinâmicas de instruções (traços). Cada traço, assim como as outras abordagens, possui um contexto de entrada e saída.

O mesmo teste de reuso aqui é aplicado a fim de determinar quando um traço pode ser reusado ou não.

Diferentemente dos blocos básicos, os traços não limitam-se a ter desvios como última instrução. Pelo contrário, um único traço de instruções pode conter vários desvios. Além disso, nenhum suporte por parte do compilador é necessário, o que garante que o mecanismo suporta código legado normalmente.

Uma destas técnicas é o RST (PILLA, 2004a; PILLA et al., 2004) que além de reuso implementa previsão de valores. O RST utiliza tanto reuso de instruções como, principalmente, reuso de traços. Para isso, vale-se de dois RB, sendo um para o reuso de instruções e outro para reuso de traços. Outro ponto importante é o domínio de reuso implementado no RST: atualmente, apenas instruções inteiras são reusadas.

2.4 Previsão de Valores

A previsão de valores (*value prediction* (VP), em inglês) (LIPASTI; SHEN, 1996; GABBAY; MENDELSON, 1996; LIPASTI; WILKERSON; SHEN, 1996; LIPASTI, 1997; SODANI; SOHI, 1998a; GABBAY; MENDELSON, 1998; PARCERISA; GONZÁLEZ, 2000) é uma técnica especulativa que explora a redundância encontrada nos programas através da previsão de valores que são produzidos (resultados) ou usados (entradas) pelas instruções. Tal previsão é baseada em valores previamente calculados, que normalmente apresentam localidade temporal e espacial, de forma que seus valores futuros mantêm algum tipo de padrão.

Ao contrário do reuso de valores, que é um mecanismo puramente não-especulativo, esta técnica permite a execução de instruções sem que todos os seus operandos estejam disponíveis. Isto permite que as instruções tenham suas execuções antecipadas, desde que a previsão tenha boas chances de estar correta (o que é determinado através de mecanismos de confiança - *confidence techniques*). Entretanto, como nem todas as previsões realizadas estarão corretas, devem existir mecanismos de recuperação do contexto correspondente ao instante imediatamente anterior àquele em que ocorreu o erro de previsão. Nesse caso, as instruções devem ser reexecutadas com os operandos corretos.

É importante salientar que os mecanismos de recuperação do contexto (no caso de uma previsão incorreta ocorrer) já estão disponíveis nos processadores superescalares atuais, pois é necessário reconstituir o contexto anterior em caso de desvios condicionais previstos incorretamente. Neste caso, com algumas pequenas modificações, estes mecanismos podem também ser utilizados pela técnica de previsão de valores. A técnica de previsão de desvios, por sua vez, é também considerada como previsão de valores, visto que prevê se um desvio será ou não tomado. Entretanto, seu foco é em dependências de controle, enquanto que a previsão de valores, que é destacada nesta Seção, é voltada para lidar com dependências de dados verdadeiras.

A fim de minimizar as penalidades impostas pela previsão de valores incorretos, estes mecanismos utilizam uma tabela de classificação de instruções, cujo propósito é determinar quais instruções possuem melhores condições de terem seus resultados previstos corretamente, além de determinar qual o previsor a ser utilizado, no caso da arquitetura permitir múltiplos previsores. Tal tabela de classificação é geralmente construída com lógica de contador saturado, para que a avaliação das chances que cada instrução tem de ter seus resultados corretamente previstos não fique prejudicada caso o contador volte para zero no caso de um *overflow*, por exemplo.

As principais vantagens apresentadas pela previsão de valores podem ser assim relacionadas:

- Minimização de dependências de dados verdadeiras (SAZEIDES; SMITH, 1997), pelo fato de permitir que instruções cujos operandos ainda não estejam disponíveis sejam executadas;
- Início da execução de instruções de forma antecipada;
- Redução da latência de instruções de acesso à memória.

Diversos estudos apontam a eficácia desta técnica (GABBAY; MENDELSON, 1996; LIPASTI; WILKERSON; SHEN, 1996; GABBAY; MENDELSON, 1998), mas nenhum deles alia a previsão de valores com o reuso. A arquitetura RST (PILLA, 2004a), por sua vez, tem como principal objetivo unir essas duas técnicas e conseguir um desempenho superior ao atingido pelas duas técnicas separadamente. O Capítulo 3 apresenta de forma mais detalhada a arquitetura RST, que é o ponto de partida do presente trabalho.

Já a principal desvantagem da previsão de valores são as perdas causadas quando um operando é previsto incorretamente, pois exige-se que o contexto anterior ao erro seja restabelecido e as instruções após este ponto, reexecutadas. Quanto mais certeza se tiver a respeito da previsão de valores feita, menos penalidades são impostas ao processador e mais desempenho pode ser obtido.

A Figura 2.7 mostra o caminho que uma instrução percorre quando a previsão de valores é utilizada pelo processador. Depois que uma instrução (ou bloco de instruções) pertencente ao domínio de instruções onde a previsão de valores é permitida, é detectada, o mecanismo verifica se existem entradas que ainda não estão disponíveis. Se houver alguma entrada não disponível, a tabela de classificação é consultada e, se a previsão do valor for considerada segura, a entrada é obtida de uma tabela chamada *Value Prediction Table* (VPT). Este valor é usado como operando de instruções que podem iniciar sua execução antecipadamente, visto que não precisam esperar suas entradas estarem disponíveis, como no modelo tradicional. Quando os valores corretos tornam-se disponíveis (depois da execução de uma instrução), eles são comparados com aqueles que foram previstos. Se for constatado que a previsão foi incorreta, as instruções devem ser reexecutadas, caso

contrário, nada precisa ser feito. O mecanismo é então atualizado de acordo com o sucesso ou falha em prever determinado valor.

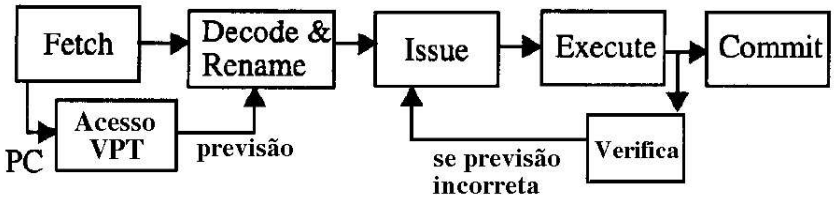

Figura 2.7: Instrução em *pipeline* com previsão de valores

Existem diversas seqüências de valores que podem ser previstas. Em (SAZEIDES; SMITH, 1997) os autores determinam três tipos: *(i)* constante, onde o valor previsto é igual ao visto anteriormente; *(ii)* variação constante, onde o valor previsto é uma diferença fixa em relação ao anterior e *(iii)* sem padrão, onde não há, pelo menos num primeiro momento, qualquer correlação entre dois valores subseqüentes. De acordo com o tipo de seqüência de valores, mecanismos focando em suas particularidades foram desenvolvidos. A seguir podem ser vistas algumas dentre as principais propostas: *last value prediction* (LIPASTI; WILKERSON; SHEN, 1996), *stride prediction* (GABBAY; MENDELSON, 1996), *trace-level prediction* (SATHE; WANG; FRANKLIN, 1998) e *context based prediction* (SAZEIDES; SMITH, 1997).

2.4.1 *Last Value Prediction*

O mecanismo de *Last Value Prediction* (LVP) (GABBAY; MENDELSON, 1996; LIPASTI; WILKERSON; SHEN, 1996) utiliza sempre o último valor previsto para uma instrução, de forma que explora seqüências constantes de valores. A Figura 2.8 mostra um exemplo de implementação para LVP: o PC é usado para endereçar a tabela de previsão, onde o valor previsto é armazenado.

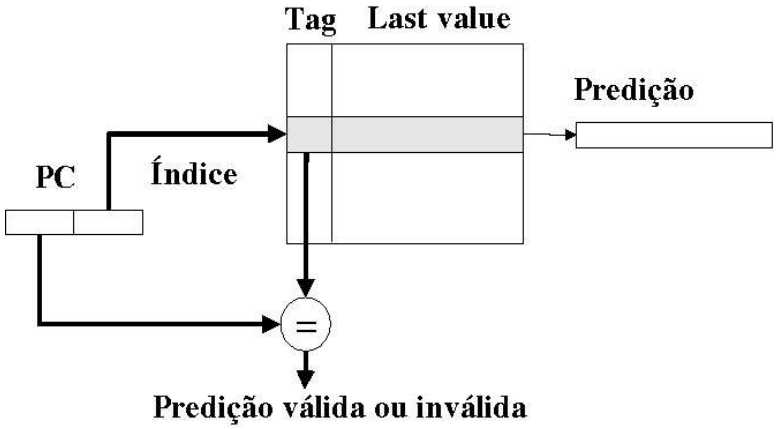

Figura 2.8: Possível implementação para LVP

Uma especialização deste mecanismo é o proposto por Gabbay e Mendelson (1996),

onde os valores são previstos para registradores. A tabela de previsão é endereçada pelo índice do registrador e não pelo endereço da instrução. Nesta técnica, a previsão está associada às instruções que escrevem/lêem em/de um mesmo registrador. A desvantagem desta abordagem é que muitas instruções manipulando o mesmo registrador podem fazer com que os valores previstos precisem ser recalculados devido ao erro de previsão. Entretanto, como vantagem, este mecanismo apresenta um tamanho de tabela pequeno, visto que depende do número de registradores da arquitetura.

2.4.2 *Stride Prediction*

Stride Prediction (GABBAY; MENDELSON, 1996) é basicamente uma extensão do LVP, de modo que agora o mecanismo permita várias instâncias de uma mesma instrução. Além disso, há um campo extra na tabela de previsão responsável por armazenar a "diferença" entre duas instâncias de instruções, que será adicionada ao último valor produzido. Isto permite que valores em laços possam ser previstos.

Na Figura 2.9 pode-se ver um exemplo de implementação do mecanismo. Através do PC, tem-se a instrução candidata a ter seu resultado previsto. Neste momento efetua-se o cálculo do valor previsto somando o último valor retornado pela instrução com a diferença entre ele e o penúltimo valor produzido.

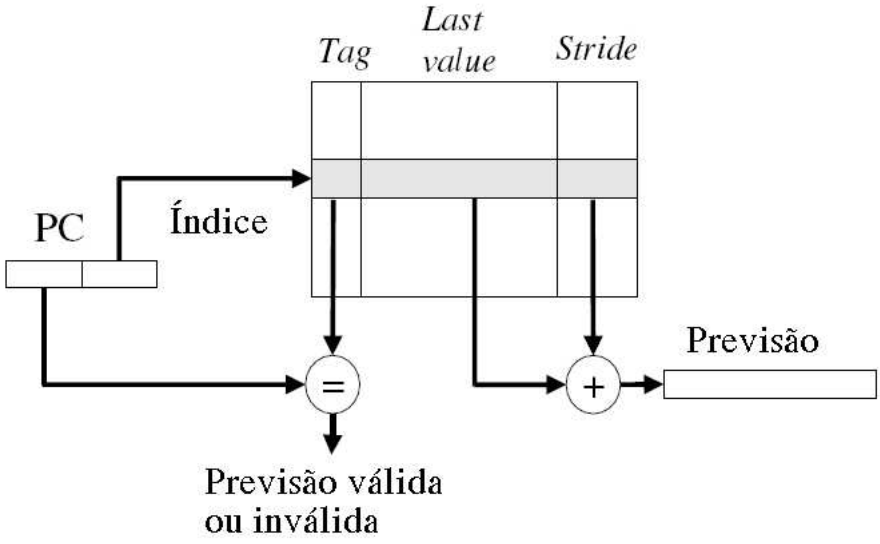

Figura 2.9: Mecanismo de *Stride Prediction*

Este tipo de preditor consegue prever comportamentos constantes ou com variação constante, como é o caso de contadores em *loops*.

2.4.3 *Context based Prediction*

O comportamento dos valores produzidos por uma instrução é utilizado pelos mecanismos baseados em contexto (*Context Based Prediction*) (SAZEIDES; SMITH, 1997; WANG; FRANKLIN, 1997). Nestes mecanismos, os últimos *n* valores observados em um

contexto são armazenados. Este modelo apresenta boa eficiência na previsão de resultados constantes, de seqüências repetitivas finitas e de estruturas periódicas não seqüenciais. Entretanto, a quantidade de valores que tem de ser armazenada para formar um histórico que permita uma correta previsão torna o seu uso proibitivo.

Uma variação deste mecanismo são os preditores **Híbridos**, que combinam outros mecanismos de previsão (REINMAN; CALDER, 1998). No caso do trabalho citado, Reinman e Calder (1998) utilizam um preditor de contexto e um *stride predictor*. Há uma comparação para determinar qual dos preditores apresenta maior probabilidade (*confidence*) de apresentar uma previsão correta. O preditor híbrido combina a habilidade do preditor contextual de reconhecer valores repetidos sem uma seqüência fixa e, também, a habilidade do *stride predictor* de prever valores ainda não vistos e que sigam uma seqüência fixa.

2.4.4 Previsão de traços

A previsão de traços, introduzida em (SATHE; WANG; FRANKLIN, 1998), produz valores para uma série de instruções pertencentes a um único traço. Entretanto, esta técnica não prevê valores intermediários ao traço, apenas as saídas do traço (*live outputs*).

Neste mecanismo o endereço do traço é usado para o endereçamento da tabela de previsão. Uma única entrada pode armazenar vários valores e, através de um campo específico, o mapeamento entre valores armazenados com suas respectivas instruções é feito.

Assim como no reuso, a abordagem em nível de traço economiza muitos recursos e a possibilidade de ganho de desempenho torna-se iminente.

2.4.5 Mecanismos de Recuperação de Contexto

Mecanismos de recuperação de contexto são essenciais em arquiteturas com previsão de valores. Como visto anteriormente, a previsão de valores não espera que todos os valores de entrada de determinada computação estejam disponíveis no momento de sua execução. Desse modo, este mecanismo utiliza valores previamente armazenados em uma tabela para supor o valor de entrada de uma computação. Entretanto, em determinados momentos, os valores calculados podem não ser os mesmos daqueles que foram previstos. Nestes casos, a arquitetura deve ser capaz de reconstituir o contexto correspondente ao instante imediatamente anterior àquele em que ocorreu o erro de previsão. Neste ponto, mecanismos de recuperação (*rollback*) fazem-se necessários.

Devido à presença de mecanismos de previsão de desvios, os processadores atuais já contam com técnicas de recuperação de contexto. Uma delas, por exemplo, consiste em se utilizar um *Re-Order Buffer* (ROB), onde as instruções serão confirmadas em ordem e se estiverem no estado “não-especulativo”. Caso um valor seja previsto incorretamente, todas as instruções subseqüentes à instrução em questão são descartadas e a busca é redi-

recionada para a próxima instrução válida, de modo que a execução seja feita com valores "não-especulativos".

Existem algumas técnicas um pouco mais complexas, como descartar apenas as instruções dependentes daquela que foi prevista incorretamente. Neste caso, o mecanismo ganha muito em complexidade e precisa de *hardware* específico para detectar dependências.

2.5 Reuso e Previsão com Acessos à Memória

Nesta Seção serão analisados alguns trabalhos relacionados ao reuso e previsão de valores com suporte a instruções de acesso à memória. Abordaremos o reuso e a previsão em subseções distintas.

2.5.1 Reuso de Valores com Suporte à Memória

Jin e Cho (2006) apresentam um estudo a respeito do reuso de valores de memória possíveis de serem explorados em programas comuns. Os estudos e análises apresentados pelos autores consideram seqüências genéricas de instruções, não levando em conta seqüências de instruções pertencentes a um traço, por exemplo. No estudo, três esquemas são utilizados:

- *Store value reuse*, onde valores utilizados por instruções *store* são reusados de forma individual;
- *Loaded value reuse*, que reusa valores de instruções de *load* – também de forma individual;
- *Macro data reuse*, onde um ou mais *loads* que fazem referência a dados próximos em memória estão a ponto de ser executados. Neste caso, quando o primeiro *load* é executado, uma porção maior de memória é buscada e, desse modo, o segundo *load* não precisa fazer novo acesso à memória, visto que seu valor já está disponível.

Utilizando uma tabela para reuso de valores de memória (MVRT, em inglês – *memory value reuse table*), os autores mostraram que aproximadamente 70% dos *loads* tiveram seus valores reusados (execuções com SPEC2k *integer* e MiBench): *(i)* 20-25% dos *loads* obtiveram o valor reusado através de *stores* executados anteriormente e, *(ii)* entre 30-40% dos *loads* obtiveram seus valores devido à execução prévia de outras instruções de *load*. Já no SPEC2k *floating-point*, o potencial de reuso de *load*-para-*load* foi de 44%.

Além deste trabalho, Bodík, Gupta e Soffa (1999) produziram um estudo quanto ao limite de reuso de instruções de acesso à memória, além de desenvolverem uma técnica de avaliação de reuso de *loads* em nível de compilação. Os resultados por eles alcançados atingem 55% de exposição de reuso de *loads* em *benchmarks* do SPEC95, sendo que este valor passa para 80% quando a técnica de compilação é utilizada.

2.5.2 Previsão de Valores com Suporte à Memória

Reinman e Calder (REINMAN; CALDER, 1998) realizaram um estudo a respeito de técnicas de previsão de *loads*, com o intuito de reduzir a latência encontrada nas instruções de acesso à memória, ainda hoje o grande gargalo dos processadores. O trabalho analisa diferentes técnicas de previsão/especulação de *loads* de forma integrada, sendo elas:

- *Dependence prediction* pode ser usada para despachar *loads* antes de todos os endereços dos *stores* serem conhecidos, além de prever de qual *store* um dado *load* depende;
- *Address prediction* é capaz de despachar um *load* sem ter seu endereço efetivamente calculado – este acaba sendo estimado;
- *Value prediction* pode ser usada para evitar a latência de reuso de *loads* e reduzir a ocorrência de *cache misses*;
- *Memory renaming* é uma técnica que encaminha imediatamente um valor gravado para uma instrução *load* que depende do valor em questão. Para isso são usados registradores e *cache* de valor, de modo que a memória não é acessada.

Nos experimentos realizados no trabalho, duas técnicas de recuperação para previsões erradas foram utilizadas: *(i)* reexecução e *(ii)* descarte das instruções. A primeira reexecuta apenas as instruções dependentes (direta ou indiretamente) do *load* previsto incorretamente. Já a segunda técnica é similar àquela usada em erros de previsão de desvios, onde todas as instruções subseqüentes ao *load* previsto incorretamente são descartadas e a busca de instruções é redirecionada para este ponto.

Os resultados (*speedups* médio sobre a arquitetura base) obtidos pelos autores, para cada uma das técnicas de previsão citadas anteriormente e de acordo com a estratégia de recuperação para previsões erradas, são mostrados na Tabela 2.1.

Tabela 2.1: *Speedup* médio para cada uma das técnicas de previsão estudadas

Técnica de Previsão	**Estratégia de recuperação para previsões erradas**	***Speedup* médio**
Dependence prediction	descarte de instruções	7%
	reexecução	10%
Address prediction	descarte de instruções	3%
	reexecução	7%
Value prediction	descarte de instruções	12%
	reexecução	23%
Memory renaming	descarte de instruções	5,6%
	reexecução	8,2%

3 RST - REUSE THROUGH SPECULATION ON TRACES

Neste Capítulo é apresentado o RST, com detalhes de seu funcionamento e especificidades de sua implementação. Na Seção 3.1 são apresentados os conceitos fundamentais do mecanismo. Logo em seguida, na Seção 3.2, é feita uma discussão acerca das tabelas de reuso, tanto de instruções como de traços. A seguir, é apresentado o funcionamento de formação dos traços de instruções na Seção 3.3 e, ainda, cada um dos estágios do *pipeline* do RST na Seção 3.4. Finalizando o Capítulo, na Seção 3.5, são apresentados vários resultados alcançados na implementação original do RST.

3.1 Introdução ao RST

O RST (*Reuse through Speculation on Traces*) (PILLA et al., 2003; PILLA, 2004a; PILLA et al., 2004), baseado no DTM (COSTA; FRANÇA; CHAVES FILHO, 2000; COSTA, 2001), é uma técnica de reuso de traços especulativa, de forma que combina tanto o reuso como a previsão de valores simultaneamente. Tal proposta prevê o uso conjunto das duas abordagens, de maneira a não se ter grande acréscimo de *hardware* se comparado a outras técnicas que utilizam as duas abordagens de forma isolada.

O reuso de valores é uma técnica não-especulativa de exploração de redundância encontrada nos programas. Esta técnica pretende obter maior desempenho através do reuso de valores previamente calculados, de forma que não sejam despendidos recursos e tempo no cálculo de uma computação já executada. Entretanto, a principal desvantagem do reuso de valores é de precisar que todos os operandos de entrada de uma instrução estejam disponíveis no momento do teste por reuso. Neste caso, muitos ciclos que poderiam ser economizados através do reuso de instruções são gastos à espera de valores de entrada.

Neste contexto, Pilla (2004a), como dito anteriormente, propõe o uso de dois mecanismos sobre instruções e traços de programas. O RST permite que os traços sejam **regularmente reutilizados**, quando todas suas entradas estão disponíveis e são iguais às entradas anteriormente armazenadas e, **especulativamente reutilizados**, quando alguns valores de entrada não estão disponíveis. Tais traços são construídos através da inclusão de instruções até que alguma instrução que não faça parte do domínio de reuso seja

encontrada (mais detalhes da construção de traços serão abordados na Seção 3.3).

Uma das principais vantagens do RST é de que esta técnica não requer tabelas adicionais para armazenar valores a serem previstos. Desta forma, o RST utiliza pouco *hardware* adicional para implementar reuso especulativo em comparação com o *hardware* exigido pelas técnicas de reuso não-especulativo.

A previsão de valores, como já visto, pode sobrecarregar os recursos da arquitetura, já que, quando um valor é previsto, várias instruções podem tornar-se prontas para execução e demandar unidades funcionais, barramento, etc. Quando os traços são reusados especulativamente no RST, suas instruções são diretamente enviadas para o estágio de confirmação, de forma que os estágios de despacho, iniciação e execução não são utilizados para o traço inteiro. Portanto, o reuso especulativo não aumenta a demanda por recursos, mesmo que previsões incorretas sejam encontradas: em alguns casos elas poderão ser reusadas.

O RST pode reusar tanto instruções como traços, mas apenas os últimos podem ser especulativamente reusados. Isto se deve ao fato de que traços possuem mais do que uma instrução e o reuso especulativo é justificado. Além disso, o reuso/previsão é feito apenas para a mesma instrução, de forma que instruções idênticas em diferentes endereços de PC não podem ser reusadas/previstas como sendo apenas uma.

No RST, o conjunto de tipos de instruções considerado para ser reusado é chamado de domínio de reuso de modo que os traços são formados com base neste conjunto. As instruções de acesso à memória não fazem parte deste domínio e, portanto, não são consideradas no RST.

3.2 Tabelas de Reuso

No RST, as instruções, depois de terem sido executadas uma vez e mostrarem potencial para reuso, são armazenadas em uma tabela chamada **Memo_Table_G**, enquanto que os traços, depois de construídos, são armazenados em uma tabela chamada **Memo_Table_T**. A Figura 3.1 mostra uma entrada na Memo_Table_G, onde não há suporte a reuso de memória.

bits	*30*	*32*	*32*	*32*	*1*	*1*	*1*
	pc	sv_1	sv_2	res/targ	jmp	brc	btk

Figura 3.1: Entrada na Memo_Table_G sem suporte a instruções de acesso à memória

Cada entrada da tabela possui basicamente sete campos, onde **pc** é o endereço de uma dada instrução; $\mathbf{sv}_1$ e $\mathbf{sv}_2$ são os operandos de entrada da instrução; **res/targ** é o resultado ou o endereço-alvo, no caso de um desvio; **jmp** especifica, quando setado, se uma instrução é um desvio incondicional; **brc** especifica, quando setado, se uma instrução é um desvio condicional; e **btk** especifica se o desvio será tomado ou não.

A Figura 3.2 mostra uma entrada na Memo_Table_T, quando instruções de acesso à memória não são permitidas nos traços. Cada entrada possui os seguintes campos: **pc** é o endereço da primeira instrução de um determinado traço; **npc** é o endereço da instrução imediatamente subseqüente ao traço; **icr** representa os nomes dos registradores de cada registro do contexto de entrada; **icv** são os valores correspondentes aos registradores do contexto de entrada; **ocr** representa os nomes dos registradores de cada registro do contexto de saída; **ocv** são os valores dos registradores de saída; **bm** é um mapa de bits usado para indicar desvios dentro de um traço; e **btk** especifica a direção dos desvios dentro do traço.

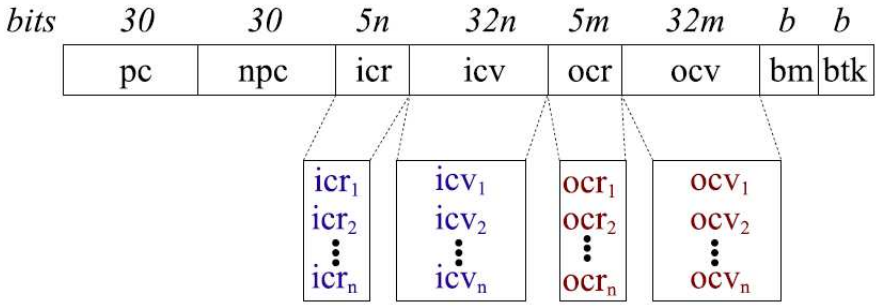

Figura 3.2: Entrada na Memo_Table_T sem suporte a reuso de memória

O endereço da instrução posterior a um traço é armazenado no *npc*, de modo que, quando um traço é reusado, o *npc* é usado para atualizar PC e não executar, assim, as instruções pertencentes àquele traço que foi reusado. Os contextos de entrada e saída são armazenados em dois conjuntos de variáveis (*icr, icv* e *ocr, ocv*, respectivamente). Os dois últimos campos são utilizados para atualizar o mecanismo de previsão de desvios.

Cada instrução é pesquisada em ambas tabelas. Caso seja encontrada na Memo_Table_T e o contexto de entrada corresponda aos valores correntes dos registrados, o traço que inicia com a instrução em questão será reusado. Caso a entrada não seja encontrada na Memo_Table_T, a instrução poderá ser reusada se houver entrada na tabela Memo_Table_G.

A ordem de precedência entre os tipos de reuso presentes no RST é a seguinte (do mais prioritário para o menos prioritário):

1. Reuso de traços
2. Reuso especulativo de traços
3. Reuso de instruções

3.3 Construção dos Traços de Instruções

No RST, os traços são dinamicamente construídos a partir de seqüências redundantes de instruções. Assim como no DTM (COSTA; FRANÇA; CHAVES FILHO, 2000), o

RST pode incluir nos novos traços em construção apenas instruções reusáveis encontradas na Memo_Table_G ou instruções cujas entradas são produzidas por instruções anteriores. Além disso, o RST pode incluir intruções mesmo que elas não estejam na Memo_Table_G, contanto que elas façam parte do domínio de reuso.

Existem basicamente duas formas de se construir traços de instruções no RST:

- *Reusable mode*, quando instruções que não estão presentes na Memo_Table_G podem ser colocadas em um traço;
- *Reused-only mode*, quando apenas instruções encontradas na Memo_Table_G podem fazer parte de um traço.

Assim que uma instrução pertencente ao domínio de reuso é finalizada, ela passa a ser considerada para a formação de traços. O processo de formação de um traço pode ser finalizado por uma das seguintes razões:

1. Uma instrução não redundante é encontrada e o mecanismo está operando no modo *reused-only*;
2. Uma instrução que não faz parte do domínio de reuso (*load/store*, por exemplo) é encontrada;
3. Limites de recursos são alcançados, como número de entradas ou saídas.

3.3.1 Processo de Criação de Traços

O processo de criação de traços no RST é melhor detalhado na Figura 3.3(a), onde as instruções, mesmo que ainda não reusadas, são incluídas em um traço até que seja encontrada uma instrução que não pertença ao domínio de reuso pré-estabelecido. Quando isto acontece, o traço é então armazenado na Memo_Table_T para futura referência.

Observa-se que desvios podem fazer parte dos traços, de forma que algumas dependências de controle são amenizadas através do reuso/previsão de traços de instruções. A eliminação de dependências verdadeiras se dá através da “execução” em paralelo de todas as instruções pertencentes a um mesmo traço.

A Figura 3.3 demonstra como é feita a formação dos traços utilizando a política *reusable*, que pode ser assim descrita: (*i*) à medida que instruções pertencentes ao domínio de reuso são encontradas, elas vão sendo armazenadas na *Memo_Table_T* até que uma instrução não pertencente ao domínio de reuso ou não redundante seja encontrada (Figura 3.3(a)); (*ii*) quando uma próxima execução alcança este traço, é feito um teste por reuso e se as entradas são as mesmas, as saídas são armazenadas nos registradores de saída. Caso algumas entradas sejam as mesmas e outras estejam sendo aguardadas, é feita

uma especulação com relação às entradas não disponíveis e as saídas são também atualizadas. Após isso, o endereço de busca passa a ser aquele imediatamente posterior ao do traço, como indicado na Figura 3.3(b).

Quando a execução é finalizada, os valores previstos e os valores reais são comparados. Se forem iguais, o estágio de finalização confirma as instruções. Caso contrário, o mecanismo de recuperação deve ser acionado e as instruções (tanto as que faziam parte do traço como as posteriores ao mesmo) são descartadas e a busca de instruções é redirecionada.

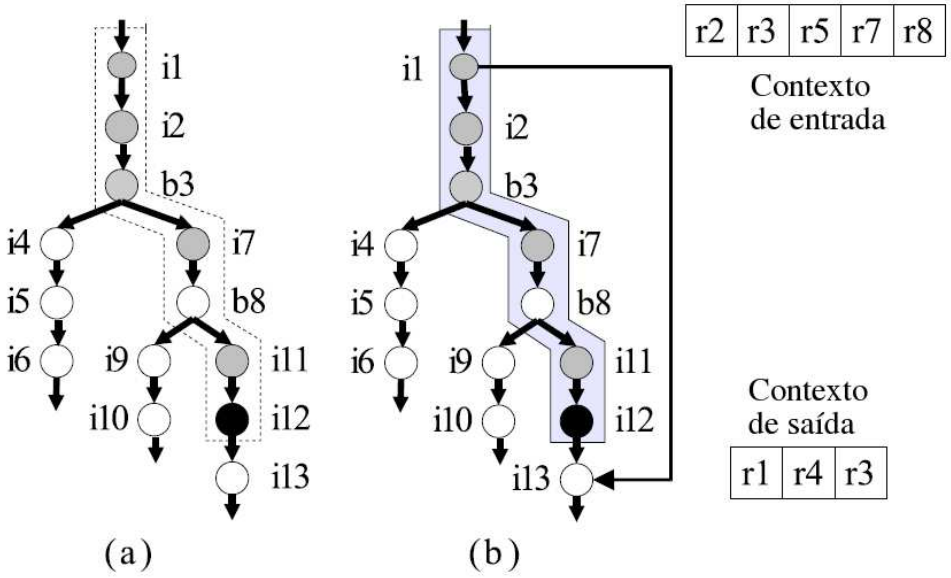

Figura 3.3: Construção dos traços usando *reusable mode*

A Figura 3.4 mostra como é feita a construção dos traços usando a política *reused-only*, também suportada pelo RST e originalmente utilizada no DTM. Em um primeiro momento, as instruções pertencentes ao domínio de reuso (na Figura, em cinza) vão sendo armazenadas na Memo_Table_G. Quando uma instrução que não faz parte do domínio de reuso (em preto) é encontrada, a busca por instruções é interrompida (como visto na Figura 3.4(a)).

Na próxima execução, estas instruções são reusadas e é formado um traço, até que uma instrução não reusada ou não redundante seja encontrada (veja Figura 3.4(b)). Este traço é então armazenado na Memo_Table_T.

Por fim, como mostrado na Figura 3.4(c), a próxima vez que esta seqüência de instruções estiver prestes a ser executada, o contexto de entrada do traço é comparado com os valores dos registradores e, então, o contexto de saída é populado com os valores corretos. Desse modo, as instruções do traço não são realmente executadas e apenas o efeito que provocam no contexto é passado adiante.

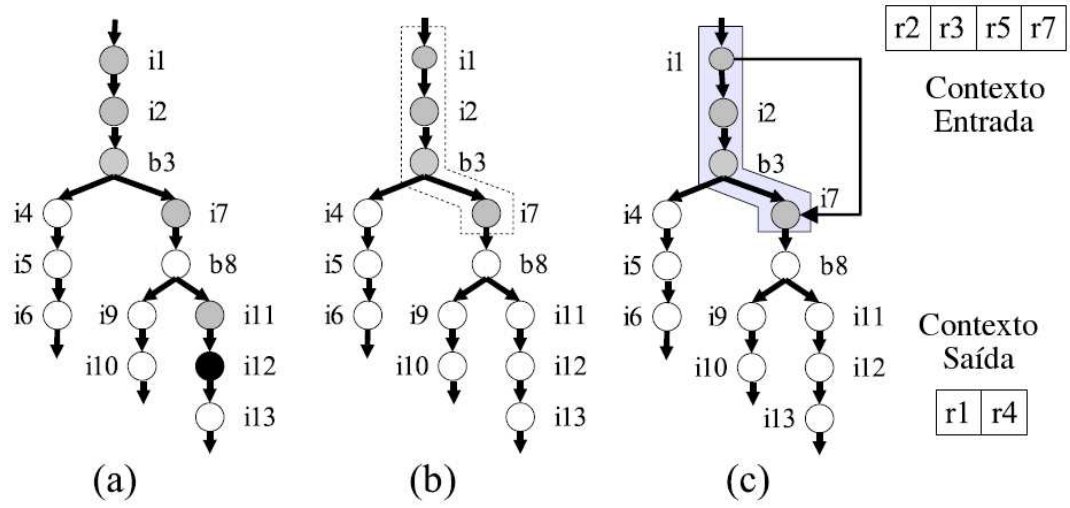

Figura 3.4: Construção dos traços usando *reused-only mode*

3.4 *Pipeline* do RST

Visto que a implementação do RST utilizou o DTM como *framework* de apoio e arquitetura base, os estágios do *pipeline* no RST também derivam e baseiam-se nos estágios do DTM. A Figura 3.5 mostra o *pipeline* do RST, onde seus quatro estágios funcionam em paralelo com o *pipeline* principal ou de instruções do processador. Desse modo, não há acréscimo no ciclo de *clock* do processador ao se introduzir o mecanismo de previsão RST.

O trabalho está dividido da seguinte maneira entre os estágios:

- O estágio **RS1** é responsável por identificar possíveis candidatos a reuso, dado um endereço do contador de programa (PC);
- O estágio **RS2** recebe candidatos a reuso e compara suas entradas com as da instrução atual. Se os valores são iguais, o RST pode reusar uma instrução e/ou traço;
- O estágio **RS3** trata do teste de erro de previsão e não tem equivalente no DTM, visto que este é um requisito imposto pela execução especulativa de valores;
- O estágio **RS4** é responsável por identificar instruções a serem gravadas na Memo_Table_G e por criar traços a partir destas instruções.

A partir de agora cada estágio será abordado de forma detalhada.

3.4.1 Estágio RS1

O estágio RS1 recebe o contador de programa (PC) da próxima instrução – enviado pelo estágio de Busca (*Fetch*) – e procura por alguma entrada nas tabelas Memo_Table_G e Memo_Table_T. A Figura 3.6 mostra a entrada (PC) e as saídas (candidatos a reuso)

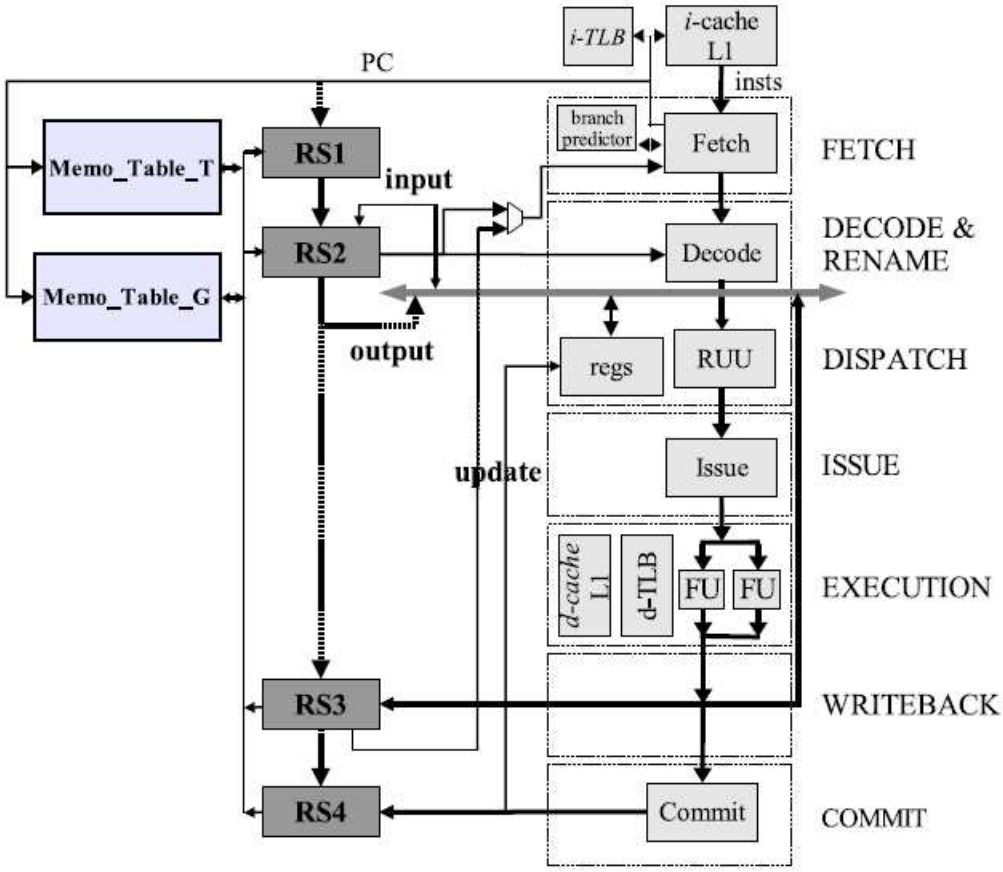

Figura 3.5: *Pipeline* do RST

deste estágio. As saídas 1 e 2 são enviadas diretamente para o estágio RS2, onde o teste por reuso é feito.

Instruções e traços que foram reusados de forma não especulativa ou que tiveram as previsões confirmadas são atualizados através das entradas 5 e 8. Já as instruções e traços que poderão ser reusados no futuro são fornecidos através das entradas 11 e 12 (vindos do estágio RS4) e armazenados logo após a consulta das tabelas.

3.4.2 Estágio RS2

O estágio RS2 é responsável pelo teste de reuso e, por isso, é um pouco diferente do estágio DS2 do DTM, já que o teste de reuso no RST deve permitir que traços com entradas desconhecidas possam ser especulativamente reusados, usando os valores previamente armazenados no contexto de entrada. Se um ou mais operandos de entrada no RS2 não são conhecidos, mas os demais operandos são idênticos aos valores atuais, um mecanismo de confiança é utilizado para determinar quando a execução especulativa pode ou não ser realizada. Caso seja feita a execução, os valores que não estão prontos são determinados com base nos mecanismos de previsão suportados pelo RST, como mostrado no Capítulo 2. No RST, este mecanismo só é acessado quando necessário.

A Figura 3.7 mostra o estágio RS2 em detalhe. Para traços candidatos a reuso, os

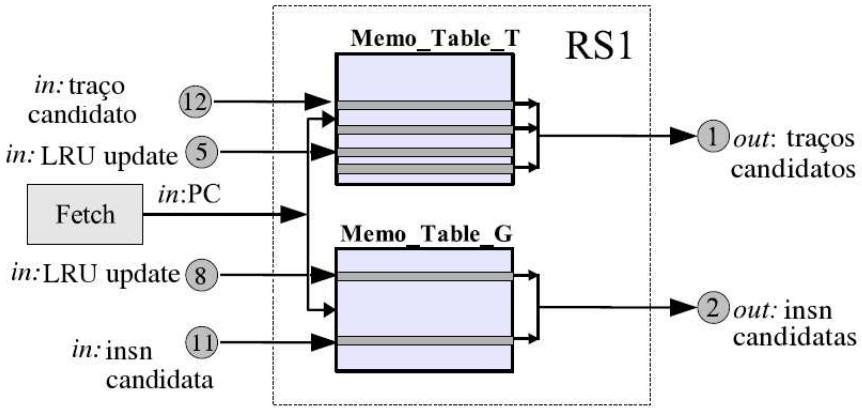

Figura 3.6: Detalhes do estágio RS1

registradores da Memo_Table_T são acessados, ao passo que para instruções, o mecanismo utiliza-se dos valores decodificados pelo *pipeline* de instruções. Depois de consultar o *register file*, os registradores são comparados com àqueles da Memo_Table_T e Memo_Table_G. As possíveis saídas para este estágio são:

1. Um traço reusado - valores de saída (saída 7), atualização da Memo_Table_T pelo método LRU (saída 5), e um novo PC (saída 6);

2. Um traço especulativamente reusado e suas entradas previstas (saídas 3, 4, 6, 7);

3. Uma instrução reusada - valor de saída (saída 7), atualização da Memo_Table_G pelo método LRU (saída 8), e um novo PC (saída 6).

Se a arquitetura não possui renomeação de registradores para tratar as dependências de valores, então cada valor no contexto de saída será uma entrada no ROB (*reorder buffer*). Instruções subseqüentes ao traço não precisarão ter acesso a valores intermediários internos ao mesmo. Tais instruções terão acesso apenas aos valores pertencentes ao contexto de saída do traço.

Entretanto, se o processador em questão tiver renomeação de registradores, o RST deve determinar quais os registradores que armazenarão os valores do contexto de saída de um traço. Como a renomeação de registradores aloca um registrador físico para cada registrador lógico, deve-se determinar as últimas escritas (antes do final do traço) nos registradores lógicos, a fim de sinalizar para o processador quais os registradores foram realmente utilizados pelo traço. Desse modo o processador saberá que pode desalocar registradores que são de uso interno do traço e, portanto, não interessam para o resultado final do mesmo. Instruções anteriores ao traço não são afetadas. Acessos subseqüentes aos registradores em questão serão tratados normalmente pelo processador.

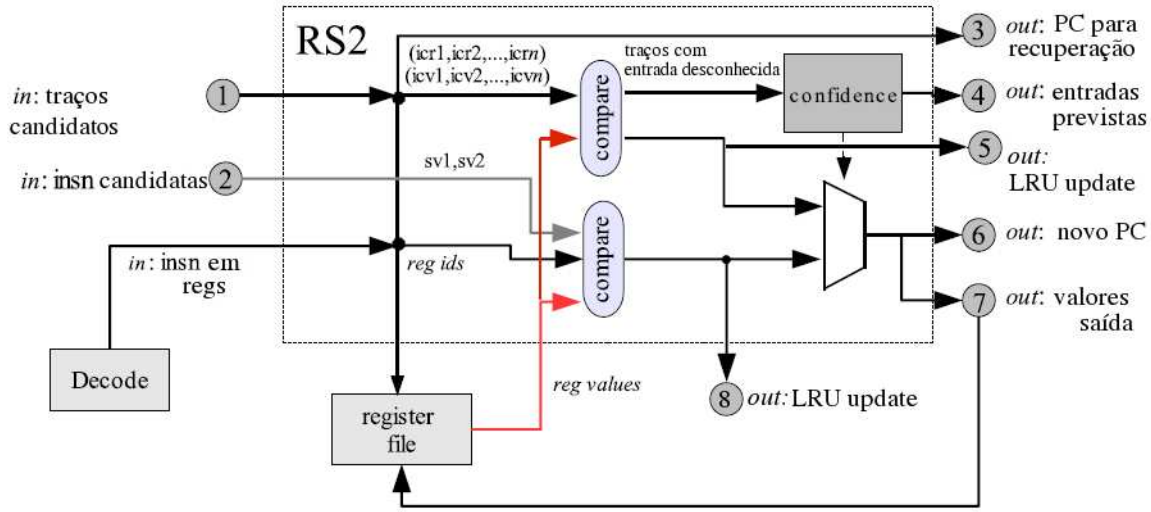

Figura 3.7: Detalhes do estágio RS2

3.4.3 Estágio RS3

A Figura 3.8 detalha o estágio RS3. Este estágio trata do teste de erro de previsão e não tem equivalente no DTM, visto que este é um requisito imposto pela execução especulativa de valores.

Este estágio compara os resultados de instruções previstas (lidos do estágio de *write-back*) com os valores reais. Se uma discrepância é encontrada, a busca é redirecionada para o início do traço e todas as instruções após o traço são descartadas. O funcionamento é o mesmo dos mecanismos usados em erros de previsão de desvios, já presentes em processadores superescalares. Instruções anteriores ao traço previsto com erro não são afetadas, pois elas não utilizam o valor que foi previsto erroneamente.

Os dados que vêm do estágio anterior são o valor do PC para recuperação (3) – no caso de um erro ocorrer – e as entradas previstas (4) para um dado traço. As entradas previstas são compostas pelo identificador da instrução produtora no ROB, o índice do registrador e o valor propriamente dito. A *Recovery Table* (RT) utiliza o identificador da instrução como índice de seus registros.

As saídas deste estágio são o modo especulativo de um dado traço (se foi previsto incorretamente ou não, saída 9), que será enviado para o estágio de *commit* do processador; uma possível atualização da Memo_Table_T pelo método LRU (saída 5); e um possível novo PC (saída 10), a ser enviado para o estado de busca caso um erro de previsão seja detectado.

Como mais de uma entrada pode ser prevista por traço, o RST usa uma RT, onde informações sobre previsões ficam armazenadas em ordem. Dessa forma, informações sobre valores previstos para um mesmo traço estão contíguas e podem ser facilmente acessadas. Os principais campos dessa tabela são:

- identificador – no ROB – da instrução produtora do valor previsto;
- identificador – no ROB – da instrução consumidora do valor previsto;
- o valor previsto;
- apontador para a entrada do traço na Memo_Table_T;
- apontador para a tabela de confiança (quando apropriado);
- número de valores previsto no traço menos 1;
- *flag* que indica se a entrada é válida ou não.

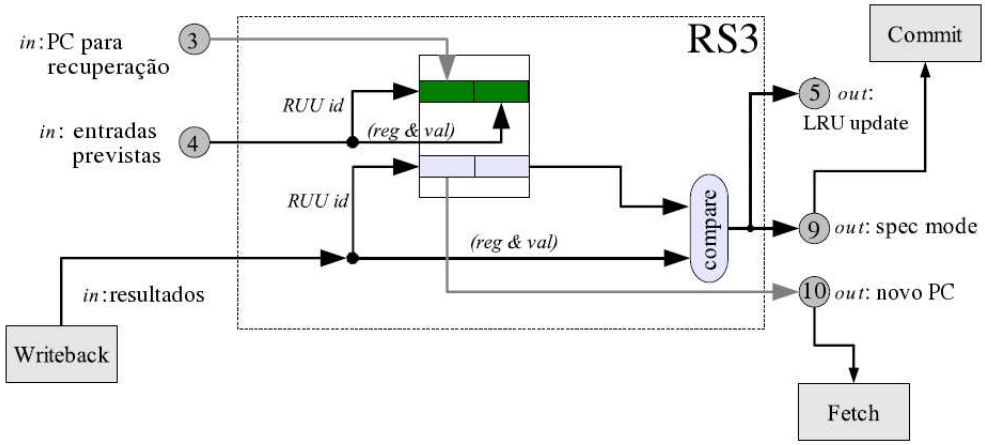

Figura 3.8: Detalhes do estágio RS3

3.4.4 Estágio RS4

O estágio RS4 é equivalente ao DS3 no DTM. Ele detecta e armazena instruções e traços redundantes. É aqui que, dependendo da política (*reused-only* ou *reusable*), os traços são formados. Quando uma previsão feita de forma incorreta é detectada, todas as instruções subseqüentes àquele traço são descartadas neste estágio e, o fluxo de execução é redirecionado para a primeira instrução que compunha o traço.

O estágio RS4 possui dois *buffers* para armazenar os contextos de entrada e saída, além de um *buffer* para criar mapeamentos de desvios para o traço que está sendo criado – Figura 3.9. As informações provenientes do estágio de *commit* são divididas da seguinte forma:

- As entradas formam o contexto de entrada;
- As saídas formam o contexto de saída;

- E, se a instrução for um desvio, ela será usada para construir o mapeamento de desvios do traço.

As saídas deste estágio são instruções candidatas a serem reusadas (saída 11) e traços candidatos ao reuso (saída 12), que serão enviados para o estágio RS1. Caso as instruções não estejam na Memo_Table_G, serão incluídas no RS1. O mesmo acontece caso os traços ainda não façam parte da Memo_Table_T.

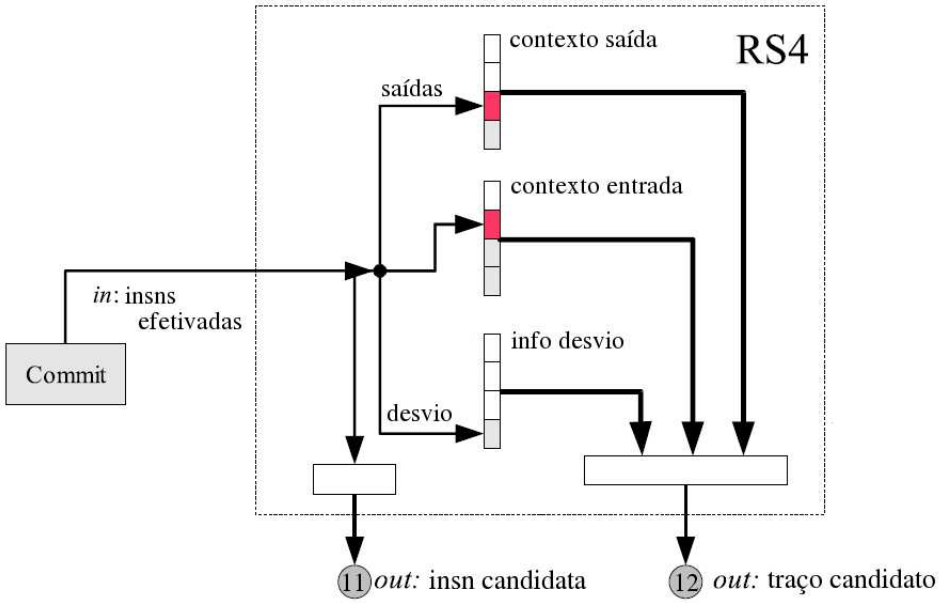

Figura 3.9: Detalhes do estágio RS4

3.5 Resultados Alcançados

Pilla (2004a; 2004; 2006) relata resultados bastante expressivos para o RST. Considerando uma arquitetura com um mecanismo de confiança que utiliza contadores saturados que permitem previsões quando os contadores atingem certos limites, o RST apresentou *speedup* médio (harmônico) sobre a arquitetura base de 1,30 (Figura 3.10). Além disso, em uma comparação com o DTM, o *speedup* médio foi de 1,075 (Figura 3.11).

Além disso, Pilla (2004a) mostrou que o tamanho dos traços formados no RST foram, em média, de 3,17 instruções por traço, enquanto que o DTM apresentou 2,32 instruções por traço (PILLA, 2004a). Isso representa um aumento de aproximadamente 37% no número de instruções e, potencialmente, aumenta a possibilidade de reuso da arquitetura.

Um outro aspecto importante foi o número de instruções de desvio presente em cada traço. Em média, o RST apresentou 58% mais desvios do que o DTM. Quando um traço que possui uma instrução de desvio é reusado, dependências de controle são eliminadas.

Um importante resultado publicado por Pilla (2004a) diz respeito à principal razão causadora para o término da formação de um traço. Como pode-se ver na Figura 3.12,

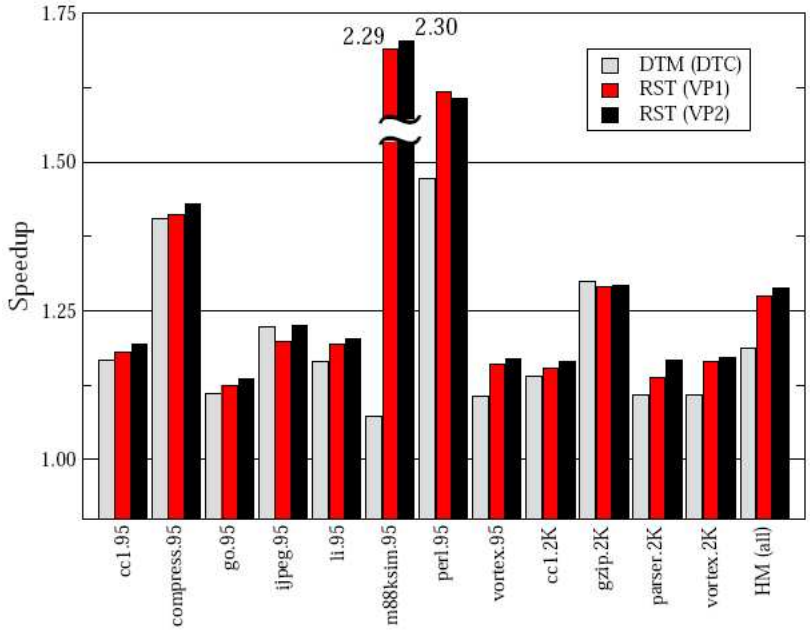

Figura 3.10: *Speedup* do RST e DTM em relação à arquitetura base

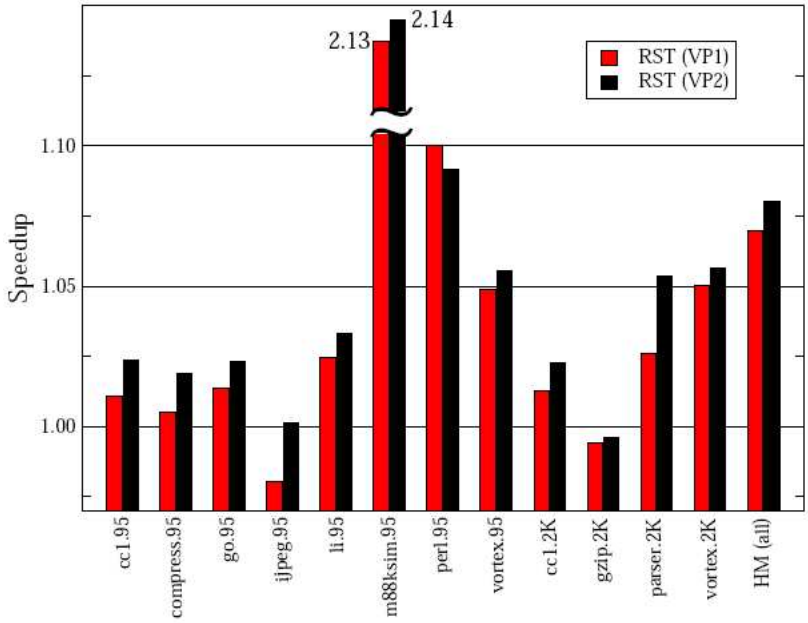

Figura 3.11: *Speedup* do RST em relação ao DTM

para todos os traços criados (ou seja, aqueles que são armazenados na Memo_Table_T ou descartados porque as instruções não estão na Memo_Table_G), as instruções de acesso à memória são responsáveis por 45% da finalização de traços. Considerando apenas aqueles traços que estão armazenados na tabela de traços, as instruções de acesso à memória representam quase que a totalidade das causas de finalização de traços.

Por esta razão, nota-se a importância da inclusão de instruções de acesso à memória no conjunto de instruções válidas do RST, visto o potencial aumento do tamanho dos traços e a minimização dos problemas de dependências de dados.

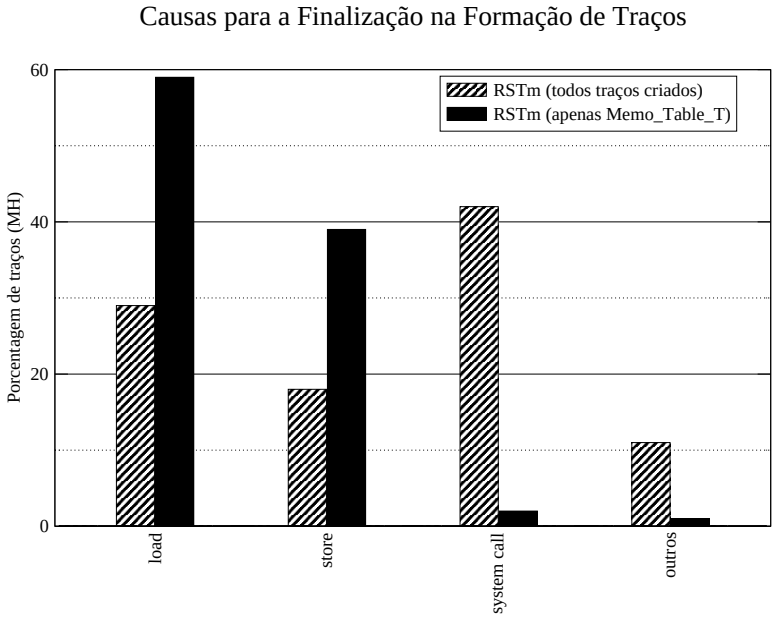

Figura 3.12: Principais causas de finalização na formação de traços no RST

4 INSTRUÇÕES DE ACESSO À MEMÓRIA NO RST

Neste Capítulo é apresentado o mecanismo proposto por este trabalho – RSTm (*Reuse through Speculation on Traces with Memory*), que altera o RST original para permitir suporte a instruções de acesso à memória no mecanismo. Aqui serão feitas considerações acerca do tema de reuso/previsão com suporte a instruções de acesso à memória (Seção 4.1), sendo seguidas por uma avaliação de todas as hipóteses analisadas, na Seção 4.2. Uma discussão sobre as escritas externas aos traços é realizada na Seção 4.3 e, em seguida, a solução proposta é apresentada na Seção 4.4. Ao final, o funcionamento do RSTm é detalhado na Seção 4.5.

4.1 Considerações Iniciais

Como já mencionado neste trabalho, experimentos realizados por Pilla (2003; 2004a; 2006) demonstram que o número médio de instruções presentes em um traço no RST é de 3,17. Além disso, tais experimentos também mostram que a principal causa para a finalização na formação de traços é a presença de instruções de acesso à memória (veja Figura 3.12 para referência), que, na implementação original do mecanismo, não fazem parte do domínio de reuso.

Um outro indício de que a inclusão de instruções de acesso à memória ao domínio de reuso é vantajosa é o trabalho desenvolvido por Vianna (2002). O autor mostra que a inclusão de um mecanismo de reuso de instruções de acesso à memória (no DTM) com antecipação de valores das instruções de leitura permite um ganho de 4,1% no desempenho médio dos *benchmarks* analisados.

Instruções que envolvem a memória são largamente encontradas durante a execução de um programa. A quantidade dessas instruções em um programa depende diretamente das características dele. Como pode-se ver na Tabela 4.1 (VIANA, 2002), a distribuição das instruções de acesso à memória, em média, é de 34% do total de instruções executadas por um programa. Dessa forma, pode-se imaginar a importância que tais tipos de instruções têm no desempenho final dos processadores.

Tal abordagem assume que instruções de acesso à memória fazem vários acessos a um

Tabela 4.1: Instruções mais executadas na arquitetura Intel X86

Tipo de Instrução	**Percentual de instruções executadas**
Load	22%
Desvio condicional	20%
Comparação	16%
Store	12%
Add	8%
And	6%
Sub	5%
Mov	4%
Call	1%
Return	1%
Outras	4%

mesmo endereço de memória. O conteúdo desse endereço de memória pode ser copiado para uma tabela e seu valor utilizado quando necessário, reduzindo, assim, a latência média de acesso à memória e a demanda por *bandwidth* no barramento, além, é claro, de eliminar as dependências de dados verdadeiras.

Entretanto, alguns cuidados devem ser tomados. Em arquiteturas que façam mapeamento de entrada e saída em memória, deve haver um mecanismo que impeça que as instruções que manipulem E/S sejam reusadas. Por isso, supõe-se o uso de um par de registradores que delimitam a área de memória em que o mapeamento é feito. Instruções específicas que habilitam e desabilitam o mecanismo de reuso também mostram-se muito úteis, especialmente na execução de trechos de código críticos, podendo ser utilizadas pelo sistema operacional quando necessário (VIANA, 2002).

A inclusão de tais instruções no domínio de reuso do RST tornará, potencialmente, os traços dinâmicos mais longos bem como minimizará o problema da indisponibilidade dos operandos de entrada. Este último problema é mais crítico no DTM, onde os traços não podem ser especulativamente executados.

A implementação de instruções de *load/store* trará custos adicionais ao mecanismo original. Entre estes custos, destacam-se o aumento no tamanho das entradas das tabelas de reuso e a necessidade de criação de mecanismos de invalidação ou antecipação de valores junto aos controles das tabelas de reuso. Este aumento no tamanho das tabelas está diretamente relacionado ao número de instruções de acesso à memória permitidas em cada traço.

Supondo-se que o suporte a instruções de memória irá demandar, de modo simplificado, dois novos campos na Memo_Table_T (veja Figura 3.2 para referência), um deles para o endereço da instrução de *load/store* (32 bits) e o outro com o valor de memória

propriamente dito (64 bits – caso seja um acesso a um tipo *double*), tem-se um grande acréscimo no tamanho da tabela e, portanto, um aumento considerável na área do processador reservada para este fim. Na próxima Seção são apresentadas algumas estratégias de implementação do mecanismo que contornam este problema.

4.2 Hipóteses Analisadas

O problema do aumento do tamanho das entradas pode ser resolvido de diversas maneiras. As principais e que merecem ser aqui detalhadas são o reuso perfeito (para efeito de simulação e determinação dos limites superiores do mecanismo), o uso de contextos de entrada e saída compartilhados, uso de uma tabela exclusiva para armazenamento de endereços de *loads* (similar à ALAT – *Advanced Load Address Table* – da Intel) (SHARANGPANI; ARORA, 2000; INTEL, ????), e, ainda, o uso de uma tabela auxiliar para armazenamento dos *loads/stores*. Veremos mais detalhes de cada uma destas propostas nas seções a seguir.

4.2.1 Reuso Perfeito

Em (LAURINO et al., 2005), foi utilizado o reuso perfeito para determinar os limites superiores de ganho de desempenho do mecanismo a ser desenvolvido. A idéia do reuso perfeito é idêntica, ou seja, permitir que *loads* façam parte dos traços de instruções. Entretanto, ao invés de implementar um mecanismo dedicado para a reutilização de *loads*, a arquitetura simplesmente testa se o valor armazenado na memória ainda é o mesmo. Dessa forma, o armazenamento do valor de memória em uma tabela de reuso passa a ser desnecessário. Essa operação de comparação é feita de forma que não influencie medidas de desempenho do mecanismo. Portanto, a cada verificação, o tempo gasto neste processo não é computado pelo simulador.

É possível notar que, para as demais instruções, o mecanismo de reuso atua normalmente, ou seja, esse mecanismo é perfeito apenas para acessos à memória. Os principais resultados apresentados no trabalho mostram que, para determinados *benchmarks*, o RST com suporte à *loads*, como mostrado na Figura 4.1, teve desempenho 3,5% superior ao RST. Já sobre a arquitetura base, os ganhos foram em torno de 28%. Além disso, um outro estudo mostrou que o ganho de desempenho é obtido pela composição de dois fatores: número de instruções reusadas e suas respectivas latências. Por isso que o RST com memória, mesmo apresentando, em média, 20,87% menos instruções reusadas do que o RST original, possui desempenho superior. Importante ressaltar também que, como esperava-se, o número de instruções pertencentes aos traços do RST com memória foi bem maior do que o RST original, tendo o primeiro mecanismo uma média de 6,7 instruções por traço, enquanto o segundo, 3,1 instruções/traço – Figura 4.2.

Por fim, uma outra análise importante mostra que a política de formação de traços

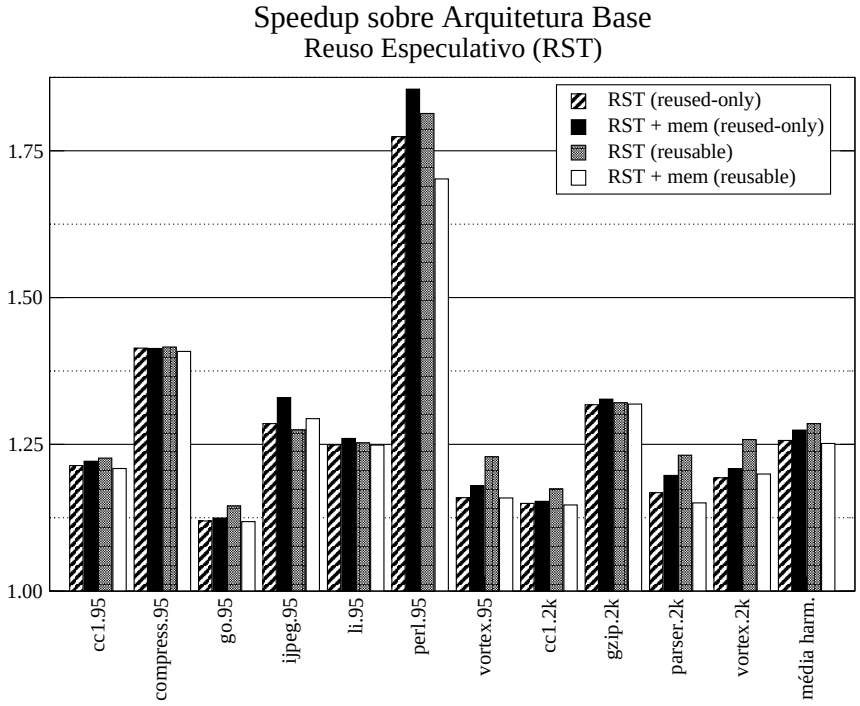

Figura 4.1: *Speedup* do RST com reuso perfeito

influencia de forma significativa o resultado final. No RST com memória, o desempenho foi melhor quando a política original (*reused-only*) para formação de traços foi utilizada.

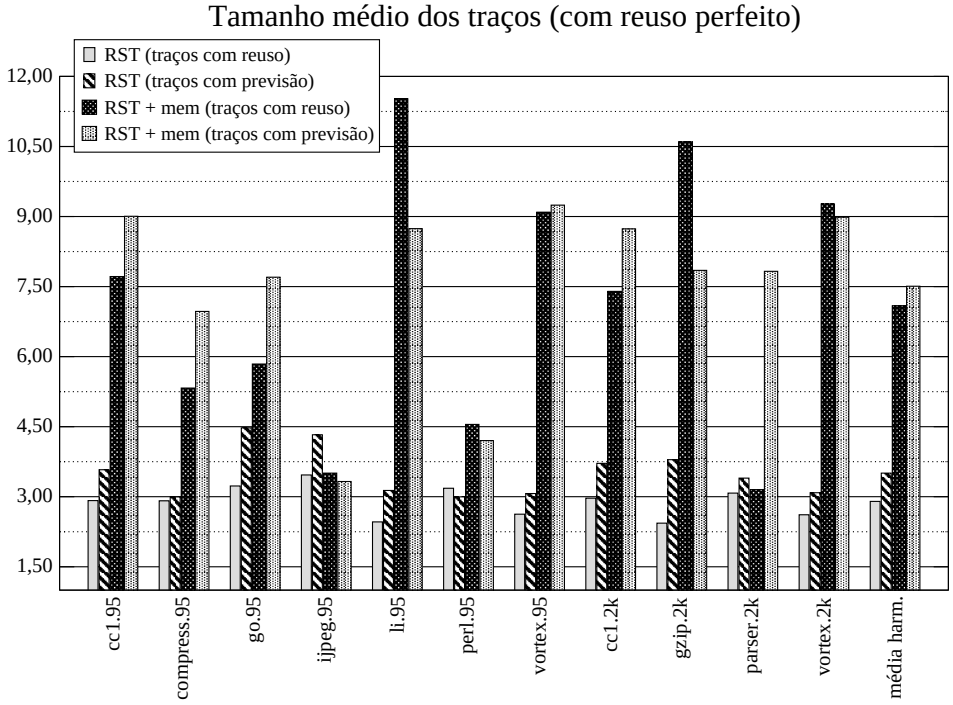

Figura 4.2: Comprimento médio dos traços com reuso perfeito

4.2.2 Compartilhamento dos Contextos de Entrada e Saída da Memo_Table_T

Outra forma de contornar o problema do aumento do tamanho das entradas é o compartilhamento dos campos da área dedicada ao contexto de entrada e saída com os endereços das instruções de acesso à memória e os valores de retorno. Supondo-se que a tabela Memo_Table_T terá que armazenar informações sobre instruções de acesso à memória presentes em cada traço, uma possível implementação seria como mostrado na Figura 4.3 (PILLA, 2004a).

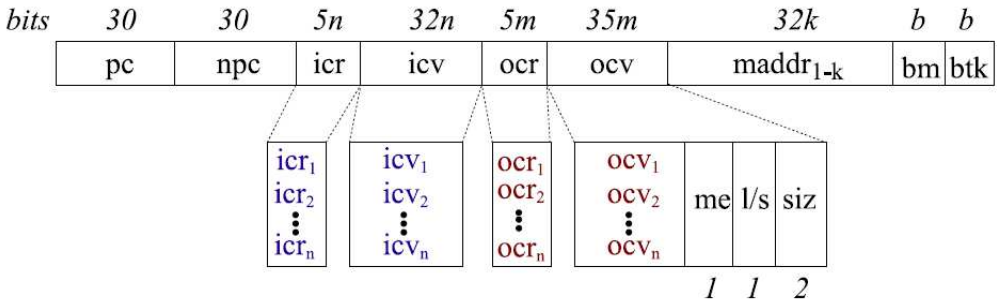

Figura 4.3: Memo_Table_T com suporte à memória

Em comparação com a Memo_Table_T sem suporte a instruções de carga/escrita – veja Figura 3.2 para referência, esta tabela apresenta os seguintes campos adicionais:

- Para cada *ocv* (valores para o contexto de saída), existem 3 campos extras:
  - *me*, indicando que os valores fazem parte de acessos à memória;
  - *l/s*, marcando se o acesso é um *load* ou um *store*;
  - *siz*, indicando o tamanho do acesso à memória (*half, single, double*).
- *maddr*, que armazenam os endereços de memória acessados pelas instruções de carga/escrita

Como pode-se notar, os valores armazenados nos endereços de memória apontados pelas instruções de *load/store* ficam armazenados no contexto de saída. Uma outra implementação desta tabela poderia utilizar o contexto de entrada para armazenar o endereço acessado pelas instruções. Entretanto, esta abordagem diminui a quantidade de operandos de entrada e saída disponíveis para um traço, causando diminuição no tamanho médio dos mesmos.

4.2.3 Uso de Tabela Exclusiva para Armazenamento de Endereços dos *Loads*

Como terceira alternativa, o mecanismo pode simplesmente não armazenar o valor de um *load*, indicando em uma tabela se os valores de memória são válidos. Portanto, esta tabela seria endereçada pelos endereços dos *loads* presentes em um traço e teria

um bit associado a cada entrada, indicando se o valor é válido ou não. O Intel Itanium utiliza essa estratégia para *loads* especulativos através da *Advanced Load Address Table* (ALAT) (SHARANGPANI; ARORA, 2000; INTEL, ????).

Neste caso, a posição da tabela que faz referência a um endereço de memória é invalidada caso ocorra um *store* entre o *load* especulativo e o *load* na sua posição original (não-especulativo). Isto seria detectado por uma instrução chamada de *load check* (*ld.c*) para casos em que apenas o *load* deve ser invalidado ou por uma instrução chamada *check-load* (*chk.a*) para casos em que mais instruções precisam ser invalidadas. Isto permite que o processador beneficie-se da grande janela de análise para determinar execuções especulativas quando apropriado.

No Itanium, a ALAT foi implementada como uma tabela de 32 entradas, com associatividade *2-way*, como pode ser visto na Figura 4.4. Cada uma das entradas tem um endereço físico que é usado para comparar com os *stores* subsequentes e determinar se há algum acesso ao mesmo endereço de memória. Além disso, cada entrada também possui alguns bits que indicam quais bytes de um endereço foram efetivamente usados pela instrução de *load* – devido a possibilidade de se ter acessos *half, single* e *double*.

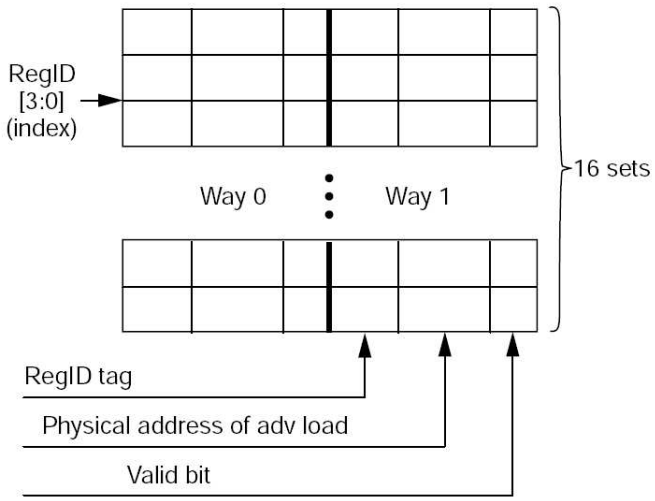

Figura 4.4: Organização da ALAT no Intel Itanium

A desvantagem desta abordagem é o fato de os valores dos *loads* não serem armazenados, visto que, como a execução das instruções de carga é feita especulativamente e de forma antecipada, o *load* é executado a cada vez que é invocado. Além disso, esta técnica necessita de apoio do compilador, o que não é o caso do trabalho proposto. Entretanto, algumas idéias interessantes aqui propostas serão levadas em consideração, como o mapa de bits que indica quais bytes de um endereço foram efetivamente utilizados pela instrução de acesso à memória.

Outros trabalhos desenvolvidos previamente também utilizam mecanismos similares para controle de predicação e execução de múltiplos fluxos (SANTOS, 2003).

4.2.4 Uso de uma Tabela Auxiliar: Memo_Table_L

A quarta maneira de contornar o problema é através da criação de uma terceira tabela, chamada *Memo_Table_L*, responsável por armazenar instruções de leitura/escrita à memória que têm potencial para ser reusadas. Desse modo, *loads* não seriam enviados para a *Memo_Table_G* nem adicionados à tabela de traços.

pc	sv1	type	maddr	valid	res
30b	32b	2b	32b	1b	32b

Figura 4.5: Exemplo de uma entrada da Memo_Table_L

A Figura 4.5 mostra uma possível configuração para as entradas da *Memo_Table_L*, que teria um número de entradas bem menor do que a Memo_Table_G, já que instruções de *load* e *store* juntas correspondem a 34% do total de instruções (veja Tabela 4.1). Por essa razão, pode-se supor que a Memo_Table_L terá $1/3$ do tamanho da Memo_Table_G. Cada uma das entradas da tabela está dividida nos seguintes campos:

- **pc**: o endereço da instrução de *load*;
- **sv1**: valor do operando fonte;
- **type**: mapa de bits que indica o tipo de acesso (*half*, *single*, *double*);
- **maddr**: armazena o endereço de acesso à memória;
- **valid**: *flag* que indica se o valor de leitura/escrita é valido;
- **res**: valor de leitura/escrita.

Neste tipo de tabela, tem-se tanto o endereço acessado pela instrução de memória como o seu valor. Além disso, este tipo de organização não limita, teoricamente, o número de *loads* e *stores* em um mesmo traço, visto que as tabelas são totalmente independentes.

4.3 Escritas Externas aos Traços

Uma das principais preocupações quando instruções de acesso à memória são tratadas é quando uma instrução de escrita externa ao traço altera o valor da posição de memória referenciada por uma instrução dentro de um traço. Duas soluções podem ser implementadas para tratar esta questão (VIANA, 2002).

A primeira alternativa é a **invalidação dos valores** de operações de leitura nas tabelas Memo_Table_G e Memo_Table_T, interceptando todas as operações de escrita na memória e verificando se há correspondência entre o endereço a ser gravado na memória e

aquele que é alvo de uma instrução de leitura presente em uma das tabelas acima mencionadas. Caso haja tal correspondência, um bit para valor de memória é trocado para inválido. Desta forma, quando é feito o despacho, este bit é consultado e, caso esteja no estado inválido, é feita a execução da operação de leitura à memória; caso contrário, a instrução seguirá o fluxo de dados normal do mecanismo de reuso.

A segunda solução, com **antecipação de valores**, tem por objetivo otimizar o mecanismo anterior e eliminar a necessidade de invalidação de operações de leitura. Assim como antes, as operações de escrita na memória devem ser interceptadas e seu endereço analisado. Se coincidir com o endereço presente em uma instrução de leitura de uma das entradas das tabelas, o valor é repassado para a mesma e não há necessidade de invalidação da operação. Entretanto, a instrução de escrita deve ser enviada para a fila de acesso à memória para que a memória seja efetivamente atualizada.

A adoção do primeiro mecanismo introduz uma complexidade adicional compatível à complexidade do mecanismo de antecipação de valores e, muito provavelmente, não trará ganhos para o RSTm.

4.4 Solução Proposta: RSTm

A implementação de instruções de acesso à memória no RST resultou em um novo mecanismo chamado RSTm (*Reuse through Speculation on Traces with Memory*) e implicou em três significativas alterações no mecanismo original. A primeira modificação diz respeito a criação de uma terceira tabela de reuso chamada Memo_Table_L, responsável pelo armazenamento e controle dos valores lidos/escritos na memória por instruções de *load*/*store*, respectivamente. A segunda foi realizada no caminho percorrido pelos traços nos estágios do *pipeline* do RST, visto que agora os traços que possuem *stores* terão um comportamento levemente diferente do que normalmente era realizado. E, por fim, a terceira modificação foi realizada na fila de acesso à memória, onde instruções do tipo *store* são interceptadas a fim de se determinar se há algum acesso a um mesmo endereço presente nas tabelas de reuso. Veremos, em detalhe, cada uma destas alterações nas subseções a seguir.

4.4.1 Nova Tabela de Reuso

Dentre todas as hipóteses analisadas, foi escolhida a utilização de uma tabela extra, denominada Memo_Table_L (Figura 4.5), que apresenta boa escalabilidade no número de *loads/stores* por traço, resolve o problema de tabelas grandes devido a necessidade de se aumentar as entradas e, também, por armazenar tanto o endereço alvo como o valor do mesmo. A nova tabela será responsável por armazenar dados das instruções de acesso à memória, terá poucas entradas em relação às outras tabelas e seu conteúdo será sempre parte do contexto de saída de um traço. Além disso, para determinar se um traço possui

alguma instrução de *load* ou *store*, basta verificar se o endereço da instrução está entre os limites do traço (*pc* e *npc* – campos presentes na Memo_Table_T). O inverso também pode ser facilmente determinado.

A fim de evitar o desperdício de espaço na Memo_Table_L, preferiu-se deixar o campo responsável pelo armazenamento do resultado da operação de memória (**res**) com 32 bits. Caso uma operação utilize um dado do tipo *double*, o mapa de bits *type* é configurado de forma apropriada e uma outra entrada na Memo_Table_L será alocada com o objetivo de armazenar o restante do valor. Portanto, um valor do tipo *double* fica armazenado em duas entradas da Memo_Table_L, como pode-se ver na Figura 4.6.

pc	sv1	type	maddr	valid	res
1000	R5	dbl	128	yes	AC
1000	R5	cont. dbl	12C	yes	9F

Figura 4.6: Exemplo de um valor do tipo *double* na Memo_Table_L

4.4.2 Alteração nos Estágios do *Pipeline*

A adição de instruções de acesso à memória no RST também exigiu algumas modificações no *pipeline* do mecanismo. Dependendo do tipo de instruções presentes no traço, este seguirá um fluxo distinto. Se instruções de escrita na memória forem encontradas em um traço no momento do despacho desta instrução, o mecanismo realizará três operações básicas:

1. Atualização do valor dos registradores do contexto de saída (como normalmente é feito);

2. Verificação da existência de alguma instrução de leitura da memória que faça referência ao mesmo endereço apontado pelo *store*. Caso o endereço seja o mesmo, o valor do store é passado para a entrada da Memo_Table_L corresponde à instrução de *load* e esta entrada é temporariamente invalidada. Só será novamente validada quando o *store* for confirmado;

3. Envio da instrução de escrita à estação de reserva e à fila de acesso à memória para manter a integridade das operações de leitura e escrita.

Caso um traço possua instruções de leitura da memória e seja candidato ao reuso, o mecanismo trata este traço normalmente, desde que as entradas na Memo_Table_L correspondentes aos *loads* em questão estejam marcadas como válidas. Ou seja, os registradores do contexto de saída são atualizados junto com o(s) registrador(es) destino da operação de leitura e são enviados ao *buffer* de reordenação para finalização.

Se estas entradas estiverem inválidas, todo o traço é considerado inválido e o reuso não é possível de ser realizado. Neste caso, a execução das instruções se dará de forma normal, seguindo todo o *pipeline* de instruções. Alguns autores (VIANA, 2002) só permitem uma instrução de leitura da memória por traço e esta deve ser a última, pois, caso os valores contidos na tabela de reuso estiverem inconsistentes em relação aos da memória, o traço inteiro não precisa ser reusado. Entretanto, as instruções de maior latência são justamente as instruções de acesso à memória, de forma que a abordagem acima não traz ganhos significativos. Desse modo, este trabalho propõe o uso de vários *loads* e um *store* por traço, tentando obter o máximo ganho com a economia de tempo que várias instruções de acesso à memória apresentam. A razão de ter-se limitado um *store* por traço deve-se, primeiramente, ao fato de que mais de uma instrução de escrita traria um aumento significativo na complexidade do mecanismo, além de aumentar a quantidade de recursos requerida para sua implementação. Além disso, dado o tamanho médio dos traços encontrados com reuso perfeito (LAURINO et al., 2005), é bem improvável que existam dois *stores* em um mesmo traço.

Estes dois novos comportamentos exigiram algumas alterações nos estágios originais do *pipeline* do RST. A seguir, detalhes das alterações realizadas em cada um deles:

4.4.2.1 Estágio RS1

A Figura 4.7 mostra como ficou organizado o primeiro estágio do RST depois das alterações realizadas. Como pode-se ver, agora o estágio de Busca envia o endereço da próxima instrução a ser executada para a Memo_Table_L onde também é realizada uma busca por instruções que estejam nela armazenada. Além disso, a partir de traços candidatos provenientes do estágio RS4, instruções de leitura e escrita são agora enviadas para a Memo_Table_L a fim de serem armazenadas (entrada 12). Como a inclusão e remoção de instruções de acesso à memória na Memo_Table_L segue a formação de traços, não é necessário atualizar as entradas desta tabela pelo LRU.

Outra modificação é a sinalização proveniente do estágio RS2 que tem por objetivo invalidar a entrada da Memo_Table_L que aponte para o mesmo endereço de uma instrução de escrita em execução.

4.4.2.2 Estágio RS2

A Figura 4.8 mostra a organização do novo segundo estágio do RST. Observando-se a figura, nota-se que foi incluído o sinal de saída 13, que corresponde à indicação de que uma instrução de escrita pretende gravar um valor diferente do último valor usado pela mesma. Dessa forma, o *store* não poderá ser reusado e deverá ser normalmente executado (envio para a *cache* de dados). Além disso, caso a Memo_Table_L tenha um *load* apontando para o mesmo endereço que este *store*, a entrada da tabela correspondente a esta instrução de leitura deve ser invalidada (pelo menos até a instrução de escrita ser

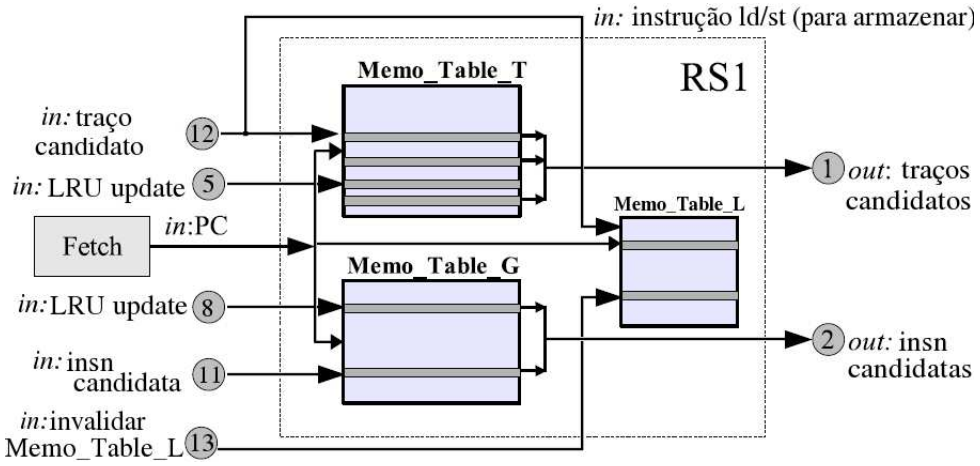

Figura 4.7: Organização do novo estágio RS1

confirmada).

A saída 13, então, servirá de entrada para o estágio RS1 que se encarregará de invalidar a entrada da Memo_Table_L.

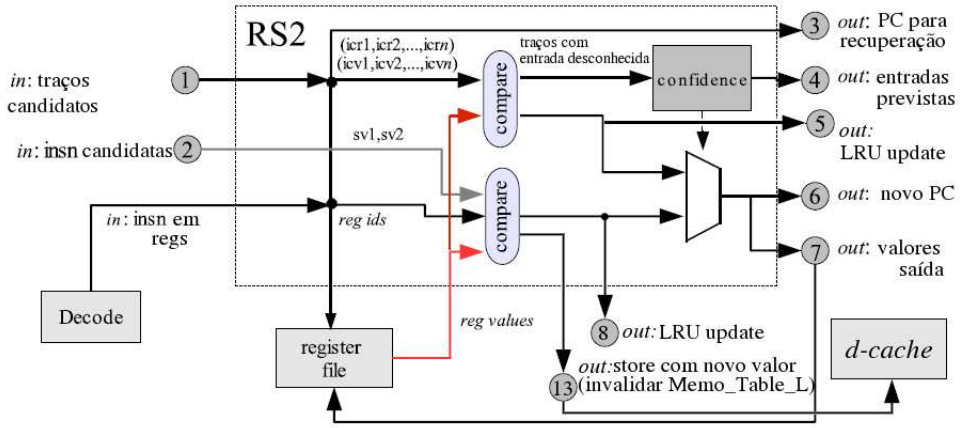

Figura 4.8: Organização do novo estágio RS2

4.4.2.3 Estágio RS3

O estágio RS3 do RSTm é idêntico ao do mecanismo original, tratando os testes de erro de previsão. Não foi necessário nenhuma alteração neste estágio, visto que os valores dos *loads* não são previstos e sim reusados.

4.4.2.4 Estágio RS4

A Figura 4.9 mostra o novo estágio RS4. Neste último, apenas uma pequena modificação no sentido de sinalizar ao RS1 que uma instrução de escrita foi terminada e que, caso exista alguma entrada na Memo_Table_L cujo endereço seja o mesmo da instrução que acaba de ser confirmada e não exista mais nenhuma instrução no ROB cujo endereço-alvo seja o mesmo, o mecanismo deve setar o bit de validade da entrada da tabela para válido.

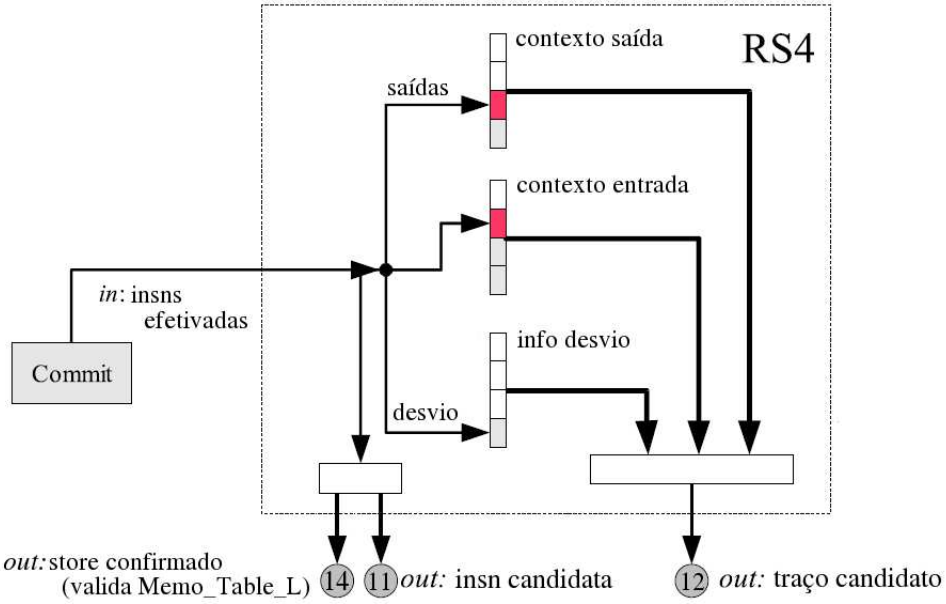

Figura 4.9: Organização do novo estágio RS4

A organização do *pipeline* depois das alterações implementadas pode ser vista na Figura 4.10.

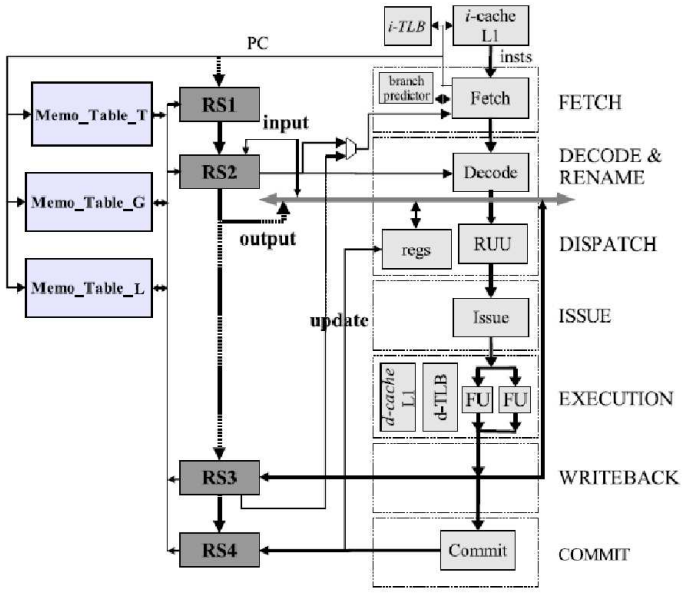

Figura 4.10: Organização do *pipeline* do RSTm

4.5 Funcionamento do RSTm

Com todas as modificações realizadas sobre o RST original mostradas anteriormente, o funcionamento geral do mecanismo proposto neste trabalho sofreu pequenas alterações. No primeiro estágio do RSTm, que funciona em paralelo com o estágio de busca do *pipeline* de instruções, as tabelas de reuso são acessadas em paralelo usando o endereço das

instruções como índice. Caso existam entradas que correspondam ao endereço procurado, estas são marcadas como candidatas ao reuso e repassadas ao estágio seguinte.

Instruções que chegam ao segundo estágio, que funciona em paralelo com o estágio de despacho de instruções, são avaliadas em busca do que poderá ser reusado. Se existe um traço selecionado e o contexto de entrada coincide com os valores atuais, o mesmo é reusado e seu contexto de saída é salvo nos respectivos registradores de saída e seus dados enviados para o *buffer* de reordenação (ROB). Caso algumas das entradas não estejam disponíveis no momento do teste, o mecanismo opta (dependendo da estratégia de confiança adotada) por prever os valores. Instruções de *load* contêm sempre o valor mais atual do endereço apontado pela instrução (garantido pelo mecanismo) se o bit de validade for válido e, para o caso de um *store*, este é comparado com àquele valor já armazenado na tabela. Caso coincida, o contexto de saída é salvo e os dados enviados para serem confirmados. Caso não existam coincidência nos valores, o traço é invalidado (pelo menos temporariamente) e as instruções são normalmente executadas.

Caso não existam traços redundantes, é verificado se existem instruções redundantes da Memo_Table_G selecionadas. Se existir alguma instrução redundante, e esta apresentar capacidade de ser reusada, o contexto de saída é salvo e a instrução enviada para o ROB.

Se nenhuma das possibilidades anteriores obtiver sucesso, determina-se se existe alguma instância da Memo_Table_L capaz de ser reusada. Havendo uma entrada redundante, o bit estando no estado válido e a instrução sendo de leitura, o contexto de saída é salvo no respectivo registrador e a instrução enviada para o ROB. Caso o bit esteja no estado inválido, a instrução é enviada à unidade funcional correspondente e à fila de acesso à memória. Caso a instrução seja de escrita, o seu valor é comparado com o armazenado na tabela. Se coincidirem, o contexto de saída é salvo e a instrução enviada para confirmação. Caso não exista a correspondência dos valores, o *store* é normalmente executado.

5 RESULTADOS

Este Capítulo apresenta os resultados obtidos nas simulações realizadas com o mecanismo proposto. Em um primeiro momento, na Seção 5.1, o ambiente em que as simulações foram feitas é descrito em detalhes. Em seguida (Seção 5.2), são apresentados resultados que demonstram os limites do mecanismo através da utilização de reuso perfeito nas simulações. Nas Seções 5.3 e 5.4 são apresentadas diversas análises e comparações a respeito do desempenho alcançado pelo RSTm, bem como características acerca dos traços formados na execução dos *benchmarks*. Além disso, algumas simulações também foram realizadas variando-se o tamanho das tabelas de reuso e a latência das memórias (na Seção 5.5).

5.1 Descrição do Ambiente de Simulação

5.1.1 Simulador

Para os experimentos realizados neste trabalho, o simulador *sim-rst* desenvolvido em (PILLA, 2004a) e baseado no SimpleScalar Tool Set versão 3.0b (BURGER; AUSTIN, 1997) foi alterado com o objetivo de descrever o mecanismo aqui proposto. Este simulador implementa um processador superescalar com profundidade de estágios no *pipeline* configurável e com o mecanismo de reuso especulativo de traços com suporte a instruções de carga na memória (RSTm).

Dentre as principais alterações, destaca-se a modificação no estágio de Busca (*Fetch*), onde instruções de escrita na memória, depois de determinado o endereço-alvo, determinam se alguma entrada da nova tabela (Memo_Table_L) deverá ser invalidada. Da mesma forma, quando estas instruções são confirmadas, a entrada correspondente deve ser validada novamente, desde que haja uma antecipação do resultado da operação de escrita na memória.

Além disso, o teste por reuso sofreu algumas alterações, visto que agora um traço contendo instruções de acesso à memória (*stores* apenas, visto que o mecanismo garante acuracidade para os *loads*) também deve validar os contextos de entrada destas instruções. Portanto, para um possível traço ser considerado como candidato ao reuso, ele deve ter

suas entradas na Memo_Table_L analisadas.

5.1.2 Metodologia de Simulação

As simulações foram feitas utilizando programas e entradas do SPEC 2000int (veja Tabela 5.1) e do SPEC 2000fp (veja Tabela 5.2) (HENNING, 2000), que são especificamente voltados para medições de desempenho de capacidade de processamento. *Benchmarks* de ponto flutuante foram escolhidos pelo fato de suas instruções serem extremamente complexas e de alta latência. Portanto, quando o reuso das mesmas for possível, a economia de recursos do processador e de tempo de execução é bastante expressiva. O objetivo na escolha dos *benchmarks* foi abranger a maior variedade possível de aplicações, de modo que a representatividade do conjunto de aplicações fosse significativa – veja as Tabelas 5.1 e 5.2 para mais detalhes de cada uma das aplicações). Todos *benchmarks* executaram no máximo 1 bilhão de instruções ou até o seu final.

Tabela 5.1: *Benchmarks* do SPEC 2000int utilizados nas simulações

Benchmark	**Descrição**
gcc	Compilador da linguagem de programação C
gzip2	Compressão
mcf	Otimização combinatória
parser	Processamento de texto
twolf	Simulador de posicionamento e roteamento
vortex	Banco de dados orientado a objetos

Tabela 5.2: *Benchmarks* do SPEC 2000fp utilizados nas simulações

Benchmark	**Descrição**
art	Reconhecimento de imagens com redes neurais
equake	Simulação de propagação de ondas sísmicas
mesa	Biblioteca de gráficos 3-D
mgrid	Multi-grid
swim	Modelagem de águas rasas

A configuração utilizada foi baseada em processadores superescalares comerciais, e, na maioria das simulações, contou com 20 estágios no *pipeline*, 128 entradas na lógica de reordenamento, 64 entradas na fila de *loads*/*stores* e três níveis de *cache*. Detalhes das configurações são descritos na Tabela 5.3.

Os três níveis de *cache* (64KB, 512KB e 2 MB) com tempos de acesso de 1, 5 e 20 ciclos, respectivamente, e memória principal com tempo de acesso de 200 ciclos são melhor descritos na Tabela 5.4.

Tabela 5.3: Principais configurações utilizadas nas simulações

Configuração	**Valor**
Fila de Busca de Instruções	16 instruções
Largura do *pipeline*	4 instruções
Previsão de desvios	em dois níveis
Branch Target Buffer (BTB)	4096 entradas, associatividade 2
Register Update Unit (RUU)	128 entradas
Load/Store Queue (LSQ)	64 entradas

Tabela 5.4: Principais configurações utilizadas nas simulações referentes às memórias

Nível de Hierarquia	**Detalhes**	**Valores**
Primeiro Nível - Instruções	latência	1 ciclo
	associatividade	4
	sets	256
	algoritmo de atualização	LRU
	tamanho total	64KB
Primeiro Nível - Dados	latência	1 ciclo
	associatividade	4
	sets	128
	algoritmo de atualização	LRU
	tamanho total	64KB
Segundo Nível	latência	5 ciclos
	associatividade	8
	sets	256
	algoritmo de atualização	LRU
	tamanho total	512KB
Terceiro Nível	latência	20 ciclos
	associatividade	8
	sets	1024
	algoritmo de atualização	LRU
	tamanho total	2MB
Memória Principal	latência (primeiro acesso)	200 ciclos
	latência (demais acessos)	20 ciclos

Além disso, o mecanismo foi simulado com as arquiteturas DTM (com reuso de valores), RST (com reuso e previsão de valores) e RSTm (suporte a instruções de memória), com duas políticas de formação de traços: *reused-only* (traços não especulativos) e *reusable* (traços especulativos). Essas configurações são mostradas na Tabela 5.5.

As tabelas de reuso foram configuradas como mostrado na Tabela 5.6. Adicionalmente, quando previsão de valores (reuso especulativo) é empregada, até dois valores

Tabela 5.5: Arquiteturas utilizadas nas simulações

Arquitetura	Política
DTM	*reused-only*
	reusable
RST	*reused-only*
	reusable
RSTm	*reused-only*
	reusable

podem ser previstos por traço.

Em (LAURINO et al., 2005) observou-se que o tamanho médio dos traços foi de 7,5 instruções, de modo que instruções de acesso à memória tendem a permitir que traços maiores sejam formados. Desse modo, os contextos de entrada e saída para a Memo_Table_T tiveram um significativo aumento de tamanho em relação ao RST original. O crescimento do limite máximo de registradores nos escopos de entrada e de saída permite que mais traços sejam formados e, eventualmente, reusados.

Tabela 5.6: Configuração das tabelas de reuso

Tabela	Parâmetro	Valor
Memo_Table_G	Entradas	2048
	Associatividade	4
Memo_Table_T	Entradas	512
	Registradores de entrada	8
	Registradores de saída	8
	Associatividade	4
Memo_Table_L	Entradas	128
	Associatividade	4

5.2 Limites do Mecanismo

Nesta seção são apresentados alguns resultados quanto aos limites de desempenho que este mecanismo pode alcançar, considerando-se a presente arquitetura. Todos os experimentos utilizam-se de reuso perfeito de acessos à memória, de modo que os custos envolvidos com operações de carga e leitura são desprezados.

A idéia do reuso é idêntica ao que foi explicado ao longo deste trabalho, entretanto, ao invés de implementar um mecanismo dedicado para a reutilização de instruções de acesso à memória, a arquitetura simplesmente testa se o valor armazenado na memória ainda é o mesmo. Dessa forma, o armazenamento do valor de memória em uma tabela de reuso

passa a ser desnecessário. É importante salientar que, para todas as demais instruções, o mecanismo de reuso atua normalmente, ou seja, esse mecanismo é perfeito apenas para acessos à memória.

A seguir será apresentada uma análise quanto aos *speedups* encontrados e logo após, uma discussão acerca de diversas características dos traços formados.

5.2.1 *Speedup*

O *speedup* do RSTm com reuso perfeito em relação ao RST original é mostrado na Figura 5.1. O eixo vertical apresenta o *speedup* alcançado, enquanto que o eixo horizontal mostra os *benchmarks* simulados. Além disso, o primeiro conjunto de barras indica a utilização da política de construção de traços original (*reused-only*), enquanto que o segundo conjunto indica a utilização da política introduzida pelo RST (*reusable*).

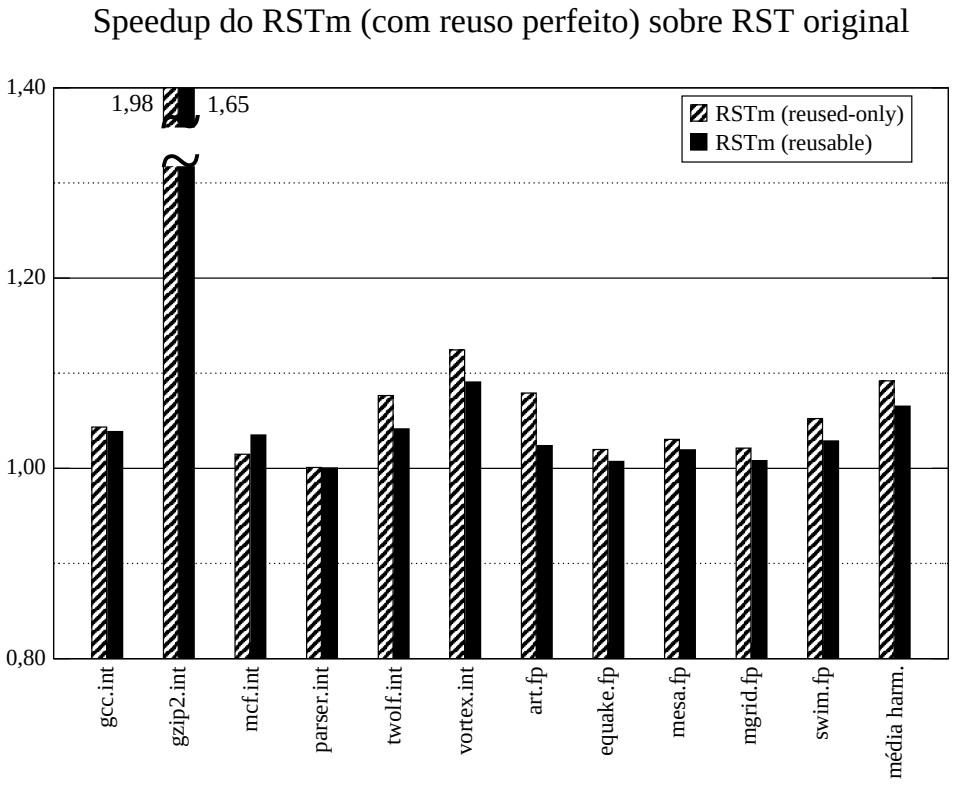

Figura 5.1: *Speedup* do RSTm (com reuso perfeito) sobre RST original

Como pode-se observar, os *benchmarks* analisados apresentaram um ganho de desempenho de $9,2\%$ para a política *reused-only* e $6,54\%$ para a política *reusable*, em média, em relação ao RST original. Para alguns *benchmarks*, como o *gzip2.int* e *vortex.int*, os resultados foram ainda mais expressivos, chegando a $98,03\%$ e $12,45\%$, em média, ambos considerando construção de traços original.

Entretanto, alguns *benchmarks* não apresentaram qualquer ganho ou apresentaram ganho muito pequeno, como no caso do *parser.int* e do *equake.fp*.

5.2.2 Composição dos Traços

Nesta seção serão apresentados basicamente três diferentes tipos de resultados: tamanho médio dos traços, percentual de intruções finalizadas nos traços e percentual de traços reusados com instruções de acesso à memória.

A Figura 5.2 mostra o tamanho médio dos traços reusados no RSTm com reuso perfeito. Além disso são mostrados resultados de traços com reuso, onde todos os operandos do contexto de entrada estão disponíveis e, também, de traços com previsão, onde até dois operandos são previstos. No gráfico, o eixo vertical mostra o número médio de instruções por traço, enquanto que o eixo horizontal mostra os *benchmarks* simulados. O tamanho médio dos traços aumentou de $3,1$ no RST original (PILLA, 2004a) para cerca de 8 no RSTm com reuso perfeito, como pode ser visto na Figura 5.2.

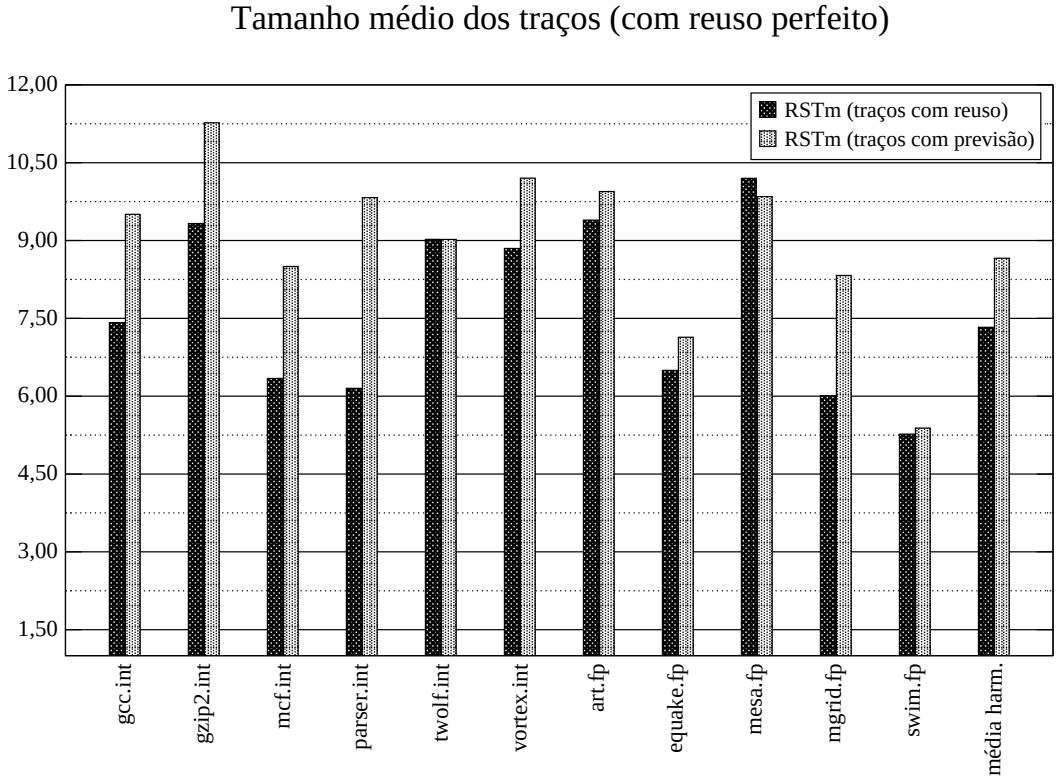

Figura 5.2: Número médio de instruções por traço (com reuso perfeito)

O percentual de instruções não executadas no RSTm, de acordo com a forma como os traços são construídos (*reused-only* e *reusable*), pode ser visto na Figura 5.3. O eixo vertical mostra o percentual de instruções finalizadas nos traços reusados, enquanto que o eixo horizontal mostra os *benchmarks* simulados. Como pode ser comprovado pelos outros resultados mostrados até aqui, nota-se uma maior efetivação de instruções quando a política original de formação de traços é utilizada. Em praticamente todos os *benchmarks* (a exceção do *twolf.int*), o percentual de instruções finalizadas foi maior com esta política, tendo obtido, em média, um total de $15,26\%$ de efetivação ao passo que com a

política mais agressiva, o percentual foi de $9,84\%$, em média.

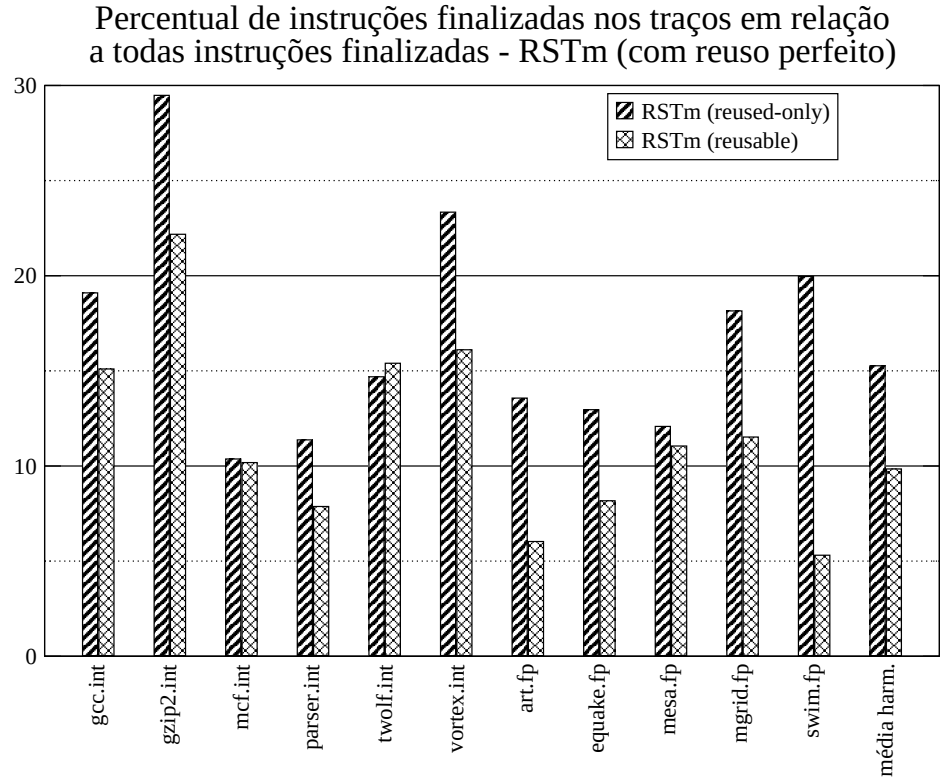

Figura 5.3: Percentual de instruções finalizadas nos traços do RSTm (com reuso perfeito)

5.3 *Speedup*

As Figuras 5.4 e 5.5 mostram o *speedup* sobre a arquitetura RST original (sem suporte à memória) e sobre o DTM (que realiza execução não-especulativa), respectivamente. O eixo vertical das Figuras apresenta o *speedup* alcançado, enquanto que o eixo horizontal mostra os *benchmarks* simulados. Na Figura 5.4, o primeiro conjunto de barras apresenta os resultados do RSTm com a política de construção de traços original (*reused-only*). O segundo conjunto mostra os resultados obtidos sobre o RST com a política de construção de traços mais agressiva, onde uma instrução não precisa ser reusada para ser candidata ao reuso (*reusable*). Os dois conjuntos de barras da Figura 5.5 apresentam os resultados para o DTM com as mesmas configurações de formação de traço utilizadas nas comparações com o RST. Além disso, uma terceira comparação é feita (Figura 5.6) em relação à arquitetura base, onde não há reuso nem previsão de instruções e/ou traços.

Em comparação com o RST original, o reuso de instruções de memória melhorou o desempenho em praticamente todos os casos, aumentando o desempenho médio (média harmônica) em $4,74\%$ no caso da construção de traços original, e de $2,17\%$, no caso da construção de traços de forma especulativa. Para casos específicos como o *benchmark*

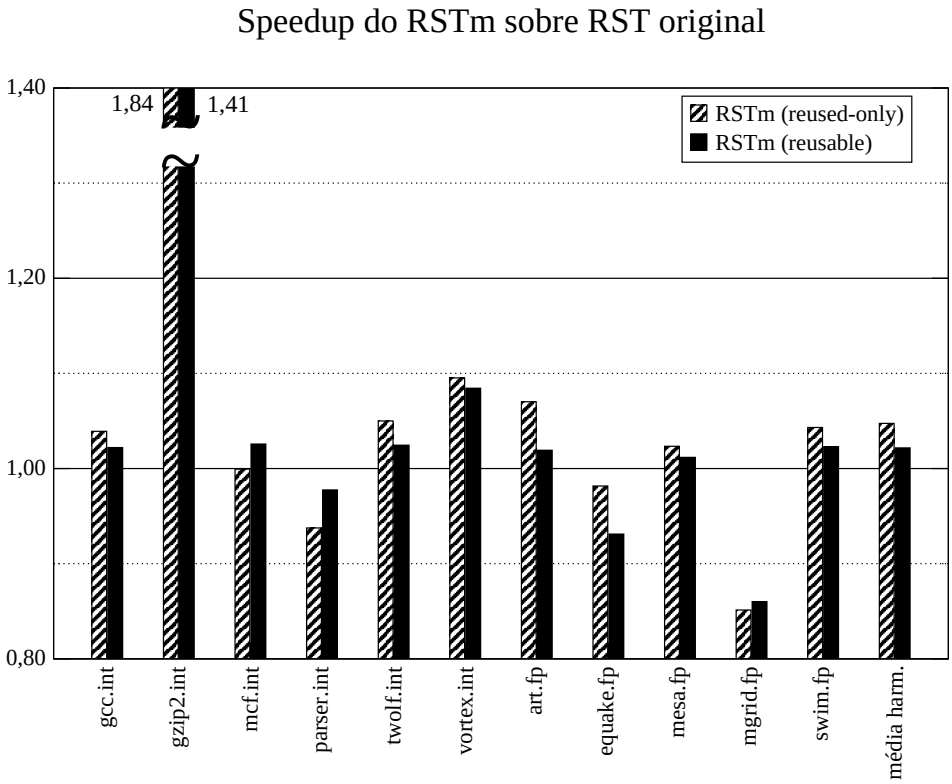

Figura 5.4: *Speedup* sobre arquitetura RST original

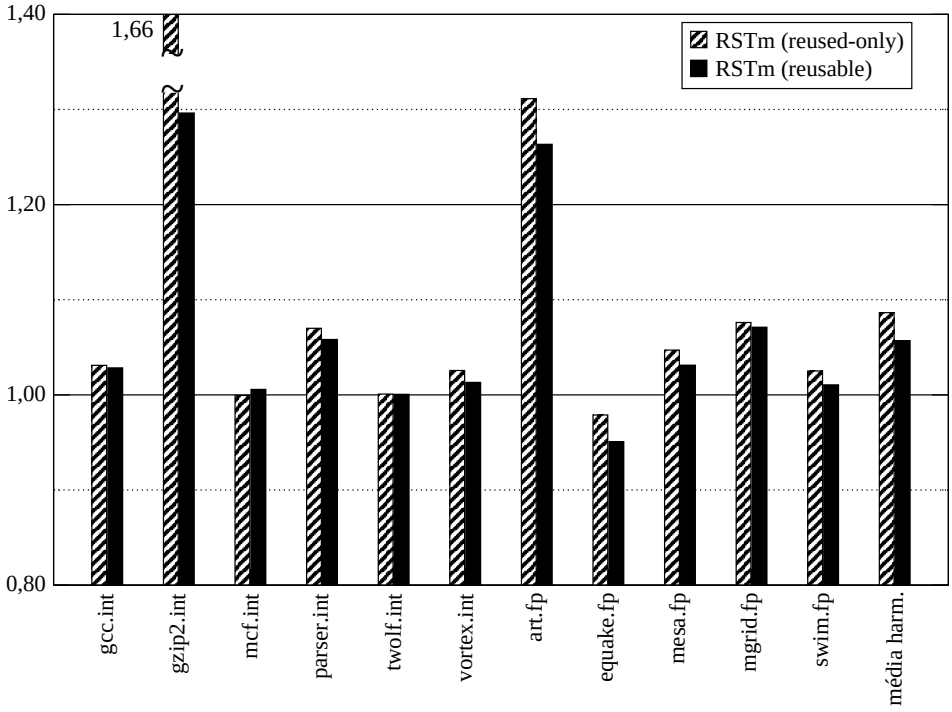

Figura 5.5: *Speedup* sobre o DTM

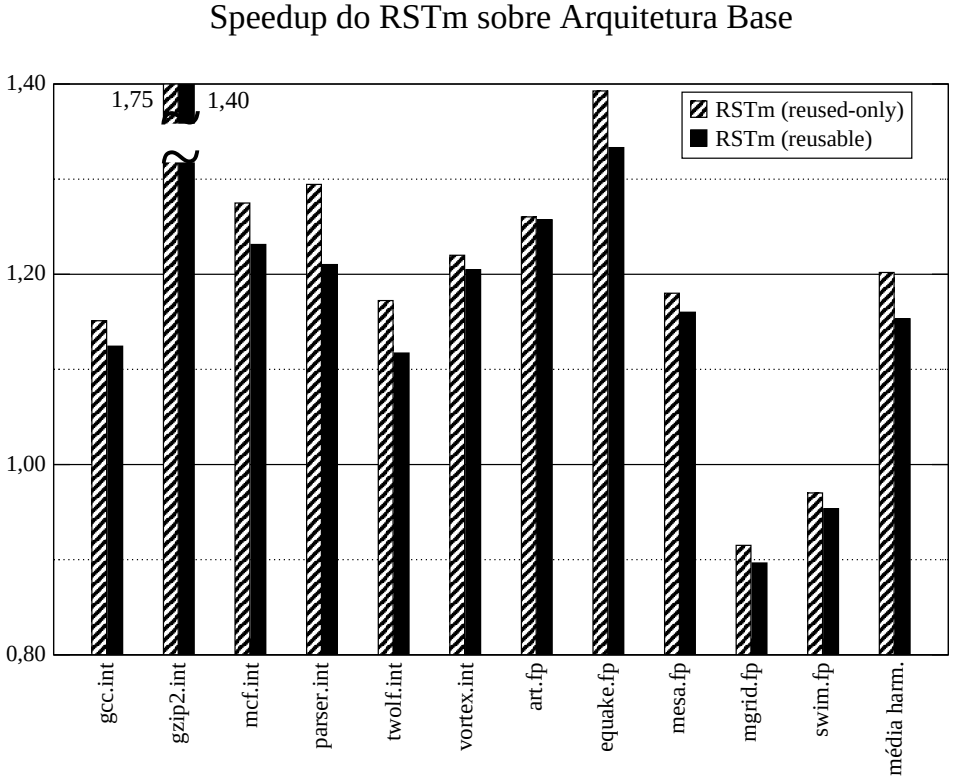

Figura 5.6: *Speedup* sobre arquitetura base

gzip2.int, a diferença a favor do mecanismo com reuso de memória é de $84,16\%$ no caso da formação original dos traços, sem traços especulativos.

Em comparação com o DTM, o mecanismo aqui proposto teve um incremento no desempenho (média harmônica) de $8,64\%$ quando construção de traços original foi utilizada e de $5,70\%$ quando utilizando construção de traços especulativa. Como já mostrado em (PILLA, 2004a), a melhor técnica de construção de traços para reuso não-especulativo segue sendo a original, determinada por Costa (COSTA, 2001).

Ao comparar-se o mecanismo com a arquitetura base, os ganhos foram ainda maiores. O *speedup* para a construção original de traços foi de $1,2019$ em média, enquanto que o *speedup* médio para a construção especulativa foi de $1,1532$. É interessante observar que seis *benchmarks* (*gzip2.int, mcf.int, parser.int, vortex.int, art.fp* e *equake.fp*) obtiveram *speedup* acima de $1,20$ para ambas as construções de traços.

Esta queda no desempenho do mecanismo, quando utilizando formação de traços mais agressiva, deve-se, basicamente, à menor probabilidade que um traço tem de ser reusado, devido a uma maior variabilidade no contexto de entrada, que agora podem conter *loads*. Da mesma forma, a política de formação de traços utilizada no DTM determina que uma instrução deve ter sido reusada antes de fazer parte de um traço, o que aumenta a garantia de que esta poderá ser reusada novamente.

Em (LAURINO et al., 2005), observou-se que os limites superiores de *speedup* para o

RST com memória usando *bechmarks* do SPEC95 na sua maioria seria algo em torno de 2%. Entretanto, alguns *benchmarks* (vide *vortex.int*, por exemplo) apresentaram desempenho superior, por dois motivos:

- maior número de *loads* permitidos nos traços, bem como contexto de entrada e saída maiores do que nos experimentos realizados em (LAURINO et al., 2005);
- todos os *benchmarks* utilizados são do SPEC2000, o que pode ter contribuído para esta alteração no desempenho de alguns deles.

5.4 Composição dos Traços

Após estudar o desempenho do mecanismo de acordo com as diferentes opções para formação de traços, é importante relacionar esse desempenho com as características que os traços reusados possuem em cada uma das configurações. Nas subseções seguintes serão apresentadas algumas destas características.

5.4.1 Tamanho Médio dos Traços

A Figura 5.7 mostra o tamanho médio dos traços reusados no DTM, RST e RSTm. Além disso, para o RST e RSTm, são mostrados resultados de traços com reuso, onde todos os operandos do contexto de entrada estão disponíveis e, também, de traços com previsão, onde até dois operandos são previstos. No gráfico, o eixo vertical mostra o número médio de instruções por traço, enquanto que o eixo horizontal mostra os *benchmarks* simulados.

O tamanho médio dos traços aumentou de $3,1$ no RST original (PILLA, 2004a) para cerca de $7,5$ no RSTm, como pode ser visto na Figura 5.7. De acordo com o que foi apresentado nas Figuras 5.4, 5.5 e 5.6, uma das razões para a melhoria de desempenho do mecanismo proposto em relação a arquitetura base é o fato de os traços apresentarem tamanho maior. Entretanto, o tamanho do traço simplesmente não implica em reuso e ganho de desempenho, visto que traços maiores tendem a apresentar um número de operandos de entrada maior, o que, potencialmente, pode significar menos redundância e menos reuso.

5.4.2 Instruções Finalizadas nos Traços

O ganho obtido pelo mecanismo proposto em termos de instruções não executadas e de acordo com a forma como os traços são construídos (*reused-only* e *reusable*) pode ser visto na Figura 5.8. O eixo vertical mostra o percentual de instruções finalizadas nos traços reusados, enquanto que o eixo horizontal mostra os *benchmarks* simulados. O gráfico mostra resultados para o RST original e RSTm. Percebe-se que o número de instruções reusadas pelo RSTm, em média, é inferior ao RST original, com ou sem traços

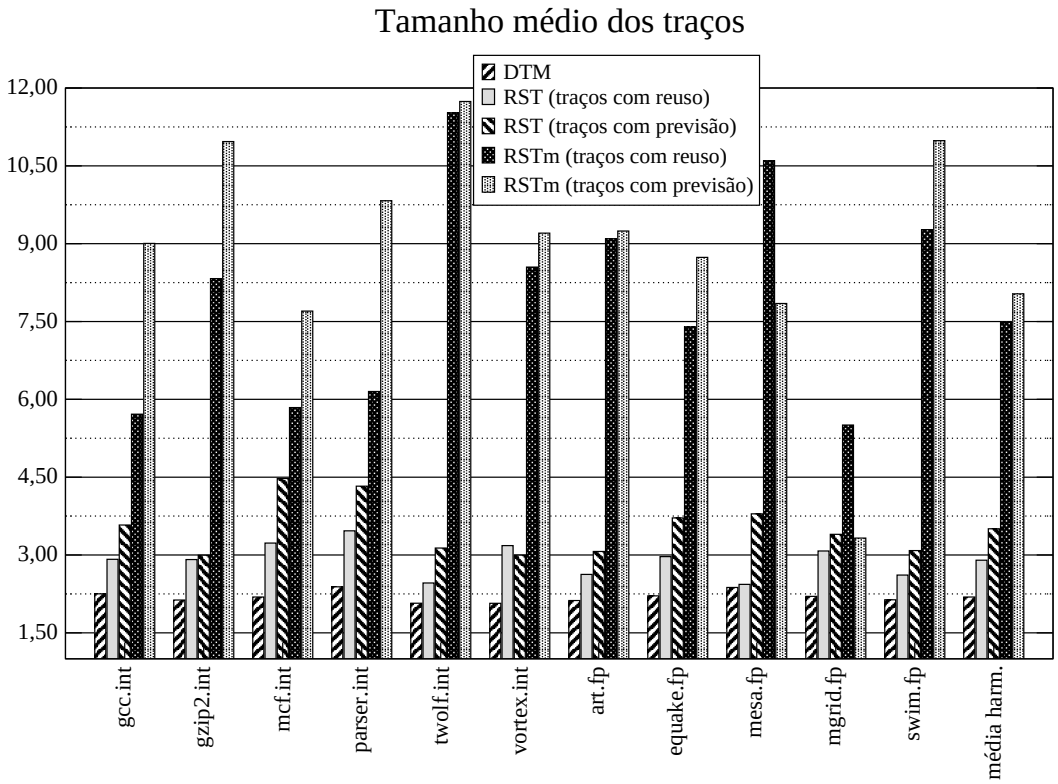

Figura 5.7: Número médio de instruções por traço

especulativos. O fato é que, ao se fazer uma correlação com o *speedup* mostrado na Figura 5.4, pode-se concluir que, mesmo reusando um número menor de instruções, o RST com memória reusa instruções com latências maiores (como é o caso dos *loads* e *stores*) e, portanto, o aumento de desempenho é justificado.

Um outro ponto interessante de ser observado é que no RSTm com política *reused-only*, o percentual de reuso alcançado é de quase o dobro quando comparando com a política *reusable*. Isto demonstra que os traços formados com a segunda política possuem pouca redundância gerando muitas penalidades à arquitetura que gasta uma parcela de tempo na recuperação do contexto anterior ao da execução do traço.

5.4.3 Traços Reusados com presença de Instruções de Memória

A Figura 5.9 apresenta o percentual de traços com a presença de instruções de acesso à memória que foram efetivamente reusados. O eixo vertical representa o percentual de traços reusados, enquanto o eixo horizontal mostra os *benchmarks* testados. O primeiro conjunto de barras é uma composição dos resultados dos traços com reuso e previsão de valores, ambos com a política original de formação de traços, para o RSTm. A seguir, o próximo conjunto é similar ao anterior, porém com a política mais agressiva de formação de traços. É possível verificar que, para a maioria dos *benchmarks*, o número de traços com instruções de carga/escrita que foram reusados no RSTm com política *reused-only* é

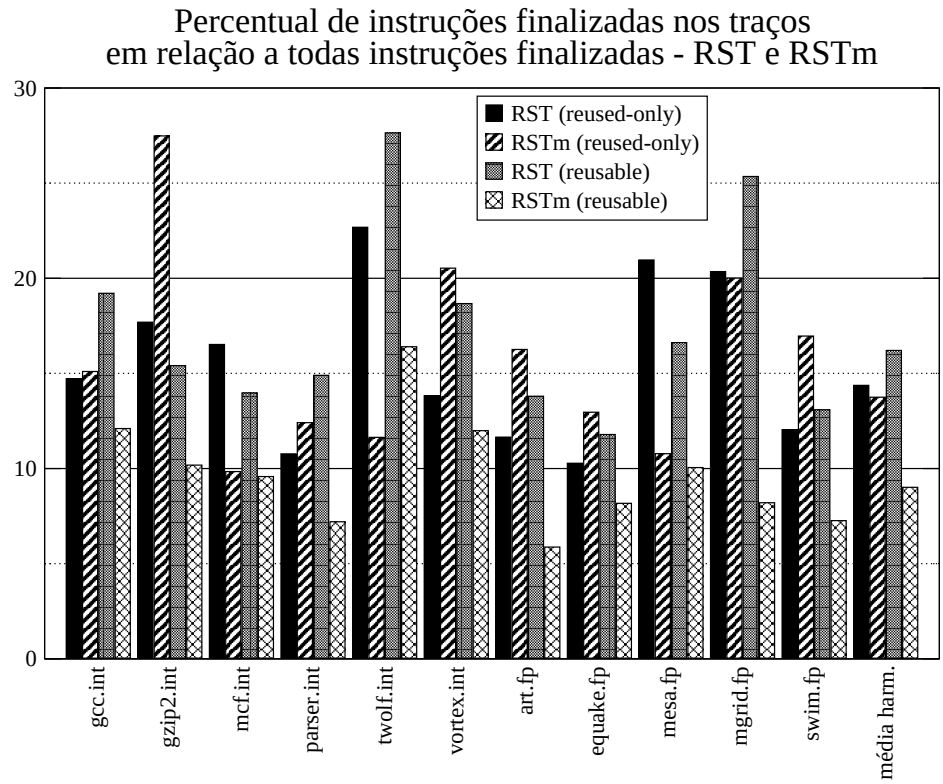

Figura 5.8: Percentual de instruções finalizadas nos traços do RST e RSTm

superior aos valores encontrados no RSTm com política *reusable*.

Além disso, para os *benchmarks twolf.int*, *equake.fp*, *mesa.fp* e *swim.fp*, entre 25 e 40% dos traços reusados, em média, possuíam instruções de carga na memória. Em casos mais específicos, como nos *benchmarks gzip2.int* e *vortex.int*, o percentual de traços reusados na arquitetura RSTm que possuíam *loads*/*stores* ultrapassou 80%.

5.5 Análises Adicionais

Nesta Seção serão apresentados alguns resultados adicionais. Estas análises são fruto de diversas simulações, variando-se basicamente o tamanho das tabelas de reuso e as latências das memórias. Ou seja, todos os demais parâmetros foram mantidos enquanto que uma destas variáveis foi alterada.

5.5.1 Variação do Tamanho das Tabelas de Reuso

Num primeiro momento, os tamanhos das tabelas Memo_Table_T e Memo_Table_L foram alterados como indicado na Tabela 5.7 (configuração *A*). Como pode ser visto, ambas tiveram seu número de entradas reduzido pela metade (em relação ao ambiente inicial de simulação), bem como a Memo_Table_T teve seus contextos de entrada e saída reduzidos de 8 registradores para apenas 6.

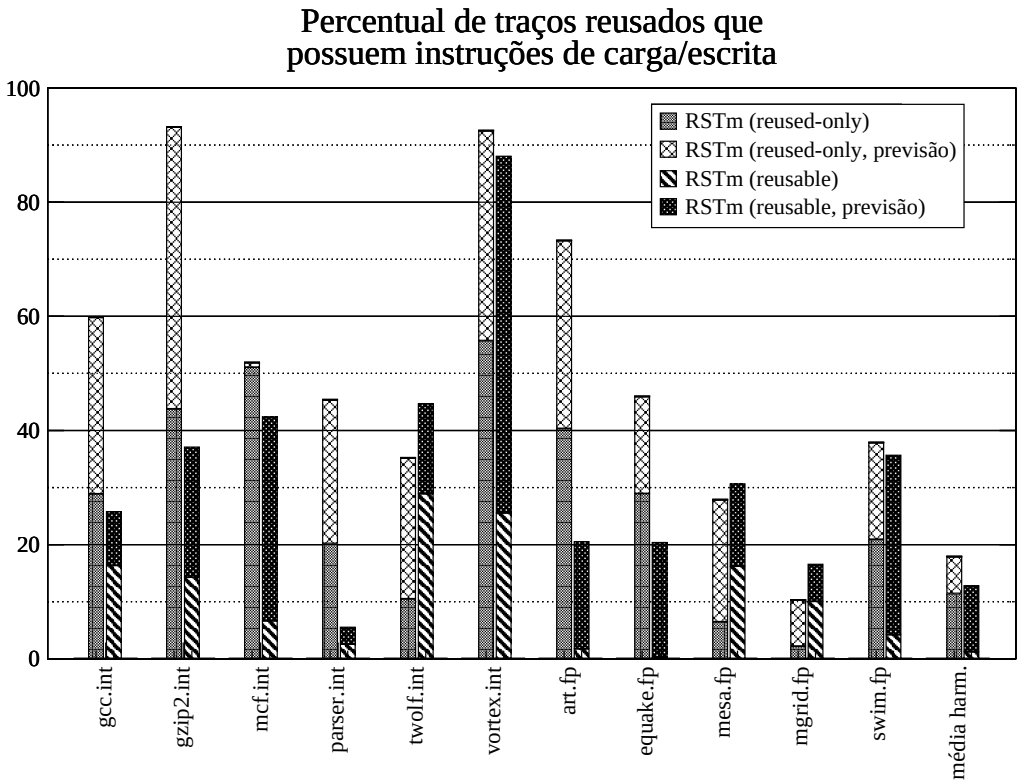

Figura 5.9: Percentual de traços reusados que contenham instruções de carga/escrita

Tabela 5.7: Configuração *A* das tabelas de reuso

Tabela	Parâmetro	Valor
Memo_Table_T	Entradas	256
	Registradores de entrada	6
	Registradores de saída	6
Memo_Table_L	Entradas	64

Com tal configuração, podemos observar alguns resultados nas Figuras 5.10, 5.11, 5.12.

O gráfico de *speedup* (Figura 5.10) indica uma diminuição no desempenho do mecanismo, pois, para a configuração inicialmente proposta, o mecanismo teve, em média, um *speedup* de $1,0474$ para a política *reused-only*; já para a configuração *A*, o *speedup* foi de $1,0275$ para a mesma política. Praticamente todos os *benchmarks* tiveram uma queda em seus desempenhos, mas especial destaque para *gzip2.int*, que teve seu *speedup* reduzido de $1,8416$ para $1,3616$ (*reused-only*) e de $1,4137$ para $1,1181$ (*reusable*).

Quanto ao número de instruções que compõe um traço – veja Figura 5.11, a configuração *A* mostra que os traços formados são menores do que aqueles gerados com a configuração original. Isso se deve basicamente ao número de registradores nos contextos de entrada e saída, que acaba por restringir o número total de operandos envolvidos

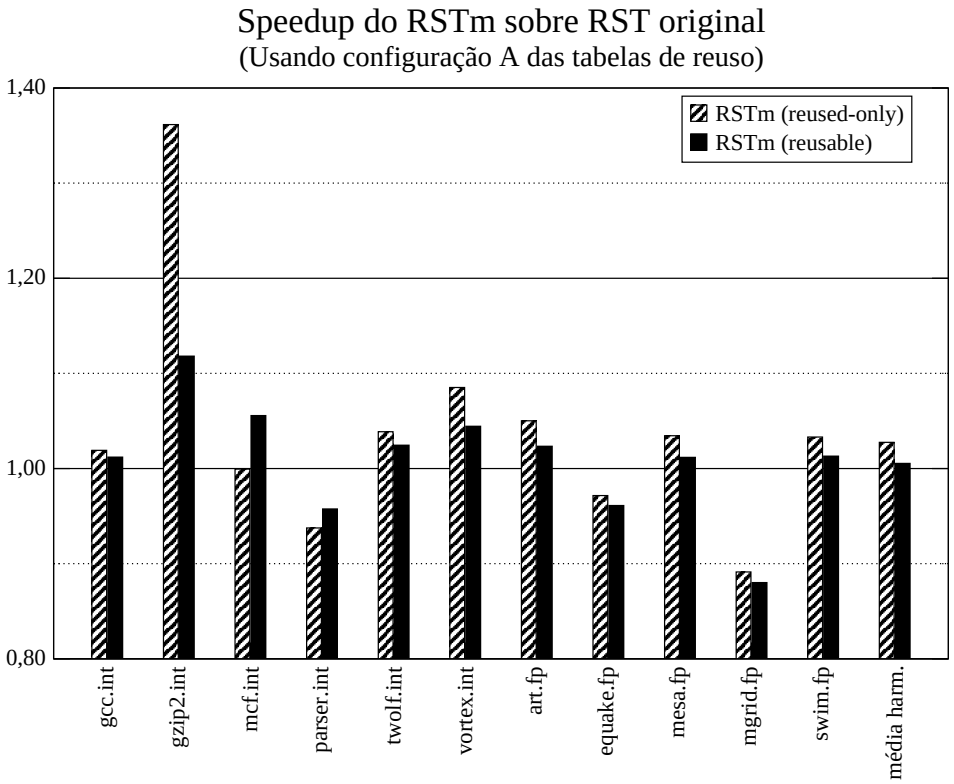

Figura 5.10: *Speedup* sobre o RST original (configuração *A* das tabelas)

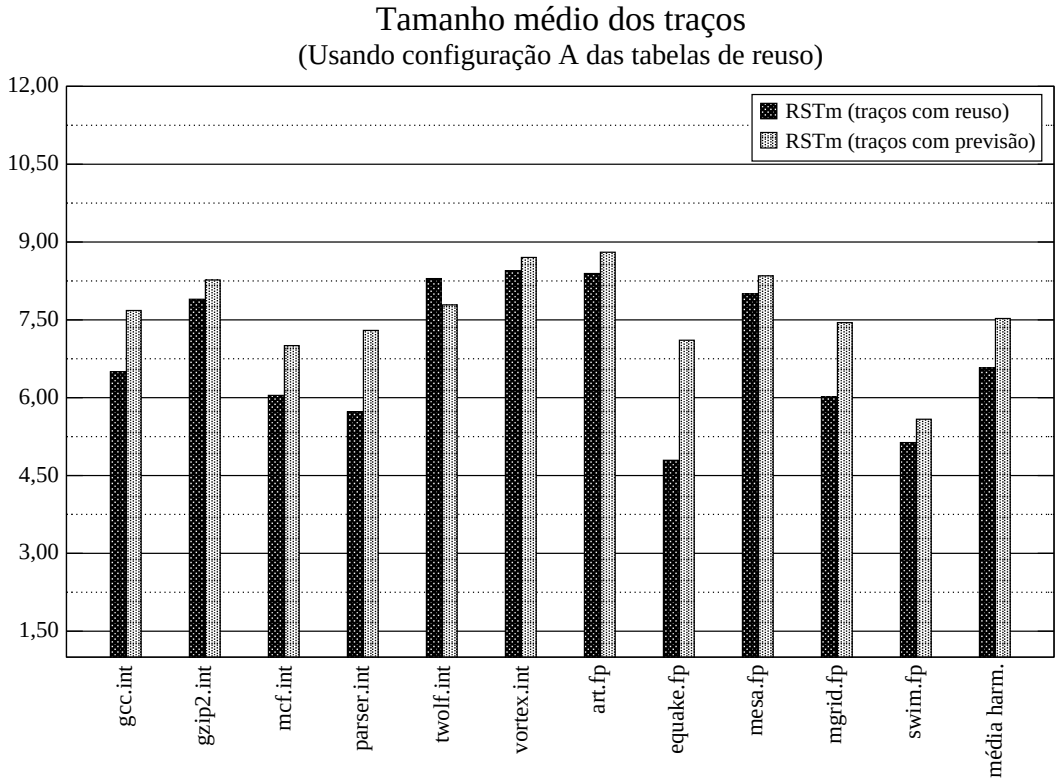

Figura 5.11: Número médio de instruções por traço (configuração *A* das tabelas)

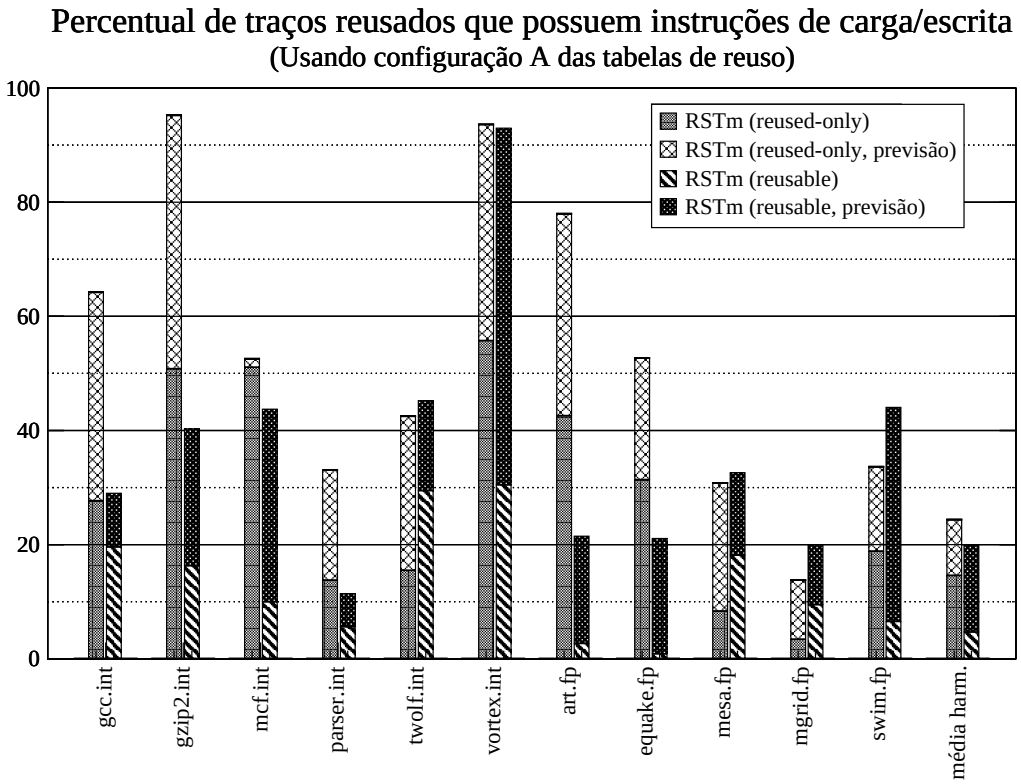

Figura 5.12: Percentual de traços reusados que contenham instruções de carga/escrita (configuração *A* das tabelas)

em um traço. Os traços formados com esta configuração têm, em média, $6,57$ e $7,52$ instruções por traço, sendo o primeiro valor para o traço com instruções onde todas as entradas já estão determinadas e o segundo, para o traço formado por algumas entradas que acabam sendo previstas. Já a configuração original apresentou $7,49$ e $8,03$ instruções por traço, respectivamente.

Ao analisar o percentual de traços que contêm instruções de carga e que tenham sido efetivamente reusados (Figura 5.12), pode-se afirmar que a quantidade de traços reusados foi maior em relação ao total de traços formados. Isto provavelmente se deve ao fato de que com traços menores, a probabilidade de os mesmos serem reusados é maior, pelo fato de terem menos variáveis no contexto de entrada. O reuso não foi maior porque o número de instruções de acesso à memória que pode ser armazenado pelo mecanismo foi reduzido à metade, de modo que instruções não tão frequentes são descartadas com mais facilidade.

Entretanto, mesmo com maior reuso, o mecanismo perdeu desempenho. Desse modo, deve-se evidenciar que há um balanceamento entre quantidade de reuso e tamanho dos traços gerados. A configuração *A* obteve reuso da ordem de $24,33\%$ (*reused-only*) e $19,89\%$ (*reusable*), em média. Já para a configuração proposta inicialmente, o reuso foi de $17,86\%$ e $12,80\%$, respectivamente.

Após a primeira modificação no tamanho das tabelas, uma nova alteração foi feita na Memo_Table_T e Memo_Table_L como indicado na Tabela 5.8 (configuração *B*). Como pode ser visto, ambas tiveram seu número de entradas duplicado (em relação ao ambiente inicial de simulação), bem como a Memo_Table_T teve seus contextos de entrada e saída aumentados de 8 para 10 registradores.

Tabela 5.8: Configuração *B* das tabelas de reuso

Tabela	**Parâmetro**	**Valor**
Memo_Table_T	Entradas	1024
	Registradores de entrada	10
	Registradores de saída	10
Memo_Table_L	Entradas	256

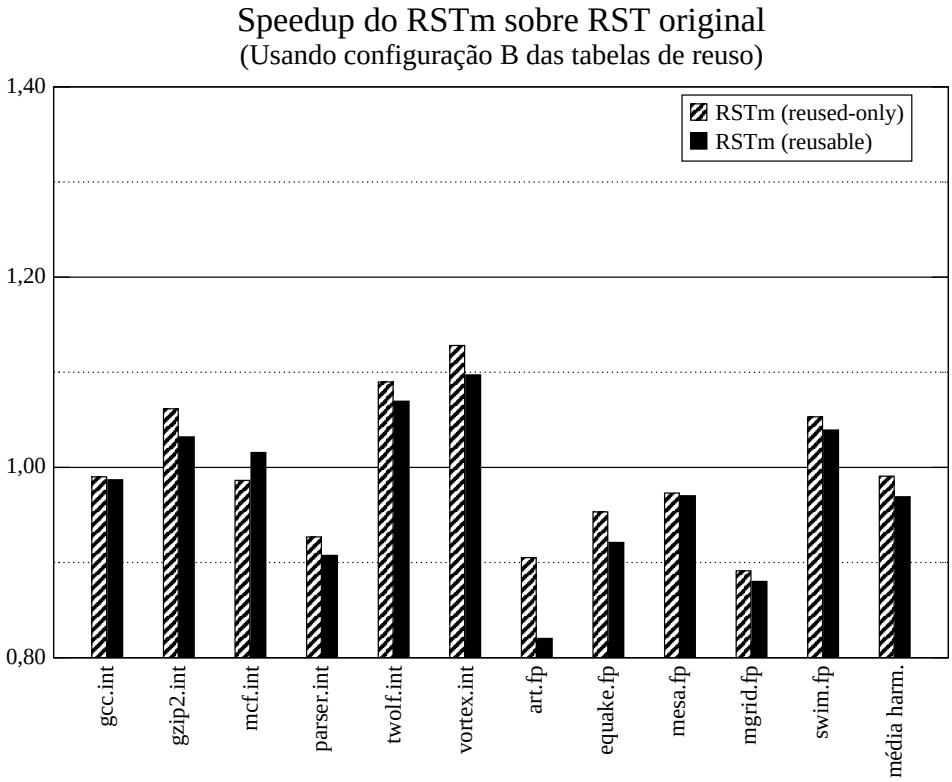

Figura 5.13: *Speedup* sobre o RST original (configuração *B* das tabelas)

Diferentemente dos resultados apresentados para a configuração anterior, o *speedup* obtido com a configuração *B* apresentou desempenho inferior ao RST original, como pode-se ver na Figura 5.13. Nesta simulação o *speedup* médio foi de $0,9908$ para construção original de traços e de $0,9692$ para a construção especulativa de traços.

Tais resultados demonstram que a configuração *B* não traz vantagens ao mecanismo, visto que subutiliza recursos alocados em demasia. Entretanto, vale ressaltar que esta

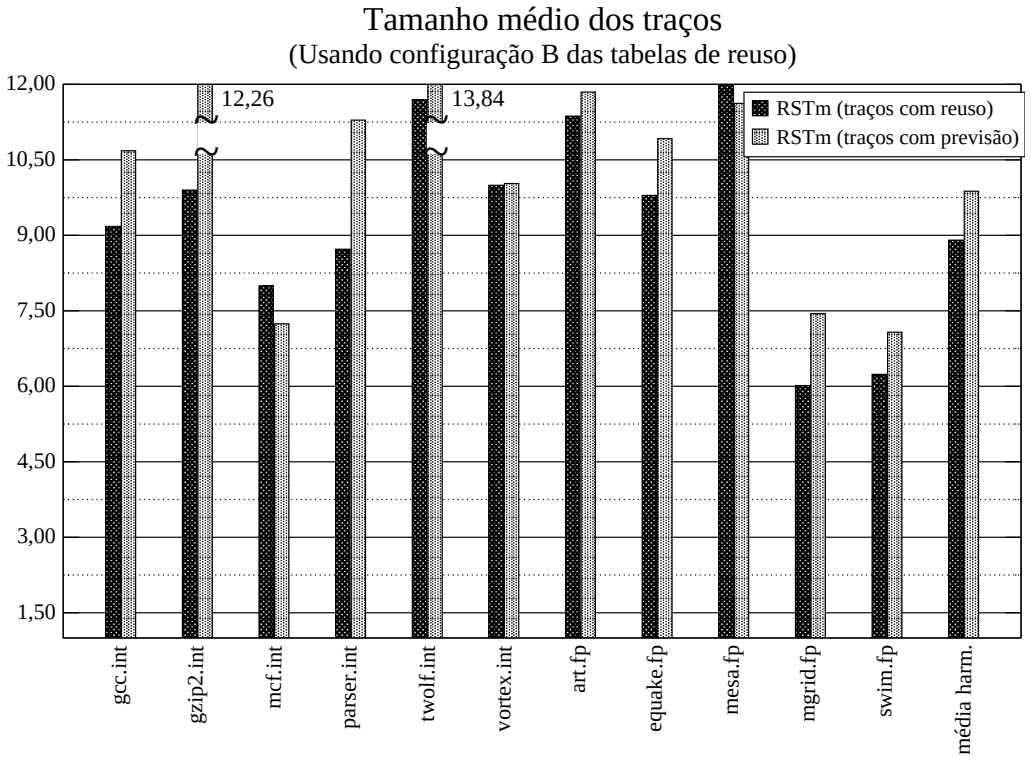

Figura 5.14: Número médio de instruções por traço (configuração *B* das tabelas)

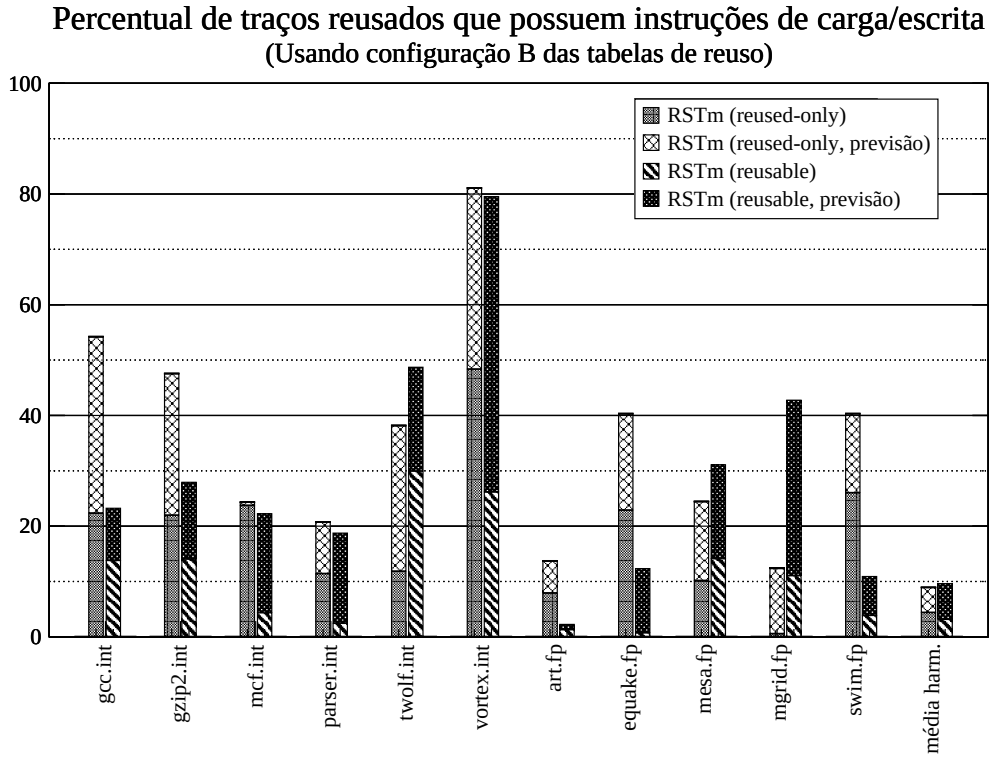

Figura 5.15: Percentual de traços reusados que contenham instruções de carga/escrita (configuração *B* das tabelas)

configuração mostrou-se adequada para quatro *benchmarks* (*twolf.int, vortex.int, mgrid.fp* e *swim.fp*), pois estes apresentam desempenho ligeiramente superior ao da configuração inicialmente proposta.

A Figura 5.14 mostra o número médio de instruções por traço (eixo y) para cada *benchmark* estudado (eixo x). A configuração *B* mostra um aumento considerável no tamanho dos traços. Para traços cujas instruções possuem entradas já determinadas, o tamanho médio foi de $8,90$ instruções, enquanto que traços com instruções cujas entradas são previstas, o tamanho médio foi de $9,87$. Como o número de operandos de entrada aumentou, a tendência de se ter traços maiores acaba sendo confirmada. Entretanto, a formação de traços maiores não refletiu de forma direta no *speedup* do mecanismo.

Como esperado, a quantidade de traços compostos por instruções de acesso à memória que foram efetivamente reusados diminuiu em relação à configuração inicial – veja Figura 5.15. Para a política original de construção de traços, o percentual médio foi de $8,9\%$, enquanto que para a política especulativa o percentual foi de $9,58\%$. Entretanto, para dois *benchmarks* em particular (*twolf.int* e *mgrid.fp*) a quantidade de reuso aumentou de forma bastante sutil.

5.5.2 Variação das Latências das Memórias

Quanto aos estudos feitos com relação ao impacto das latências das memórias no desempenho do RSTm, duas simulações distintas foram realizadas. Na primeira, a latência das memórias dos três primeiros níveis foi duplicada, como mostrado na Tabela 5.9.

Tabela 5.9: Latências das memórias para a configuração *A*

Nível de Hierarquia	**Latência**
Primeiro Nível - Dados	2 ciclos
Segundo Nível	10 ciclos
Terceiro Nível	40 ciclos
Memória Principal	200 ciclos (primeiro acesso)
	20 ciclos (demais acessos)

Após a execução dos *benchmarks* tendo as latências das memórias com a configuração *A*, o gráfico de *speedup* do RSTm em relação ao RST pode ser visto na Figura 5.16. Como esperado, o ganho de desempenho em relação ao RST original foi maior do que quando utilizando a configuração originalmente proposta, visto que o tempo de acesso a qualquer uma das memórias (exceto a memória principal) é maior na configuração *A*. Portanto, quando o mecanismo evita o acesso, a economia de tempo é maior neste caso e os ganhos médios ficaram em torno de $7,59\%$.

A segunda simulação envolvendo modificação nas latências das memórias da arquitetura está descrita na Tabela 5.10. Nesta situação, apenas a memória principal teve sua

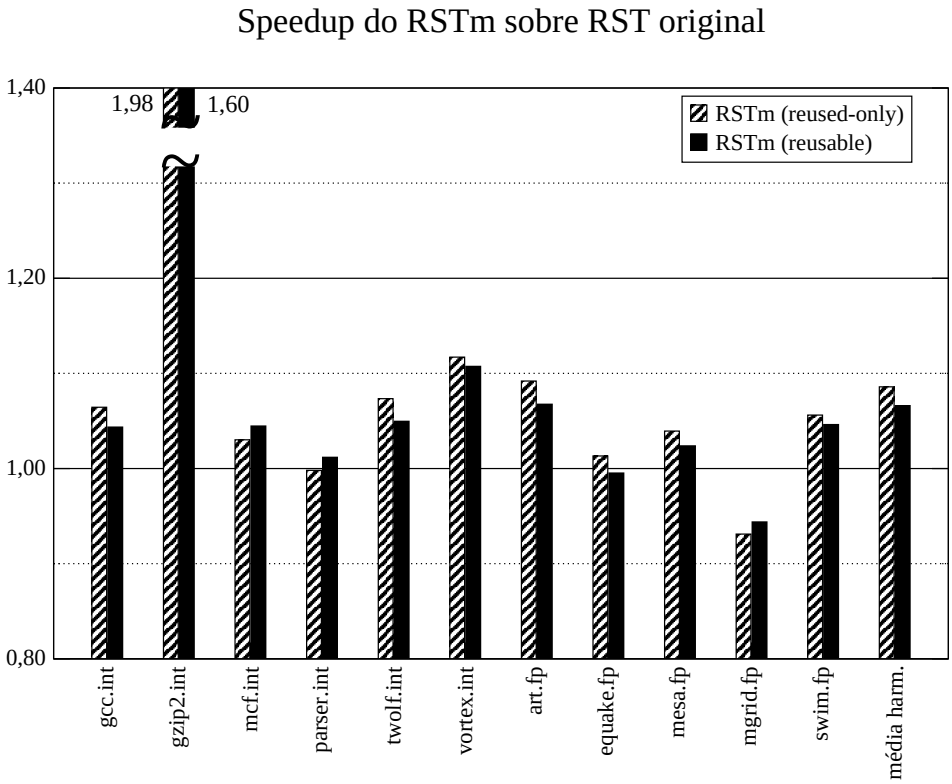

Figura 5.16: *Speedup* sobre o RST original (configuração *A* das memórias)

latência modificada de forma que seja o dobro daquela especificada na configuração original.

Tabela 5.10: Latências das memórias para a configuração *B*

Nível de Hierarquia	**Latência**
Primeiro Nível - Dados	1 ciclo
Segundo Nível	5 ciclos
Terceiro Nível	20 ciclos
Memória Principal	400 ciclos (primeiro acesso)
	40 ciclos (demais acessos)

A Figura 5.17 mostra o *speedup* do RSTm em relação ao RST original para a configuração *B*. Neste caso, ocorre também um maior ganho de desempenho em relação àquele obtido com a configuração inicial. Entretanto, o ganho obtido não foi maior do que o medido no experimento anterior, tendo sido de $5,75\%$. Tal consideração atesta que a maior parte das instruções de acesso à memória reusadas são aquelas cujos dados de entrada estão armazenados nos três primeiros níveis de memória (basicamente *caches*).

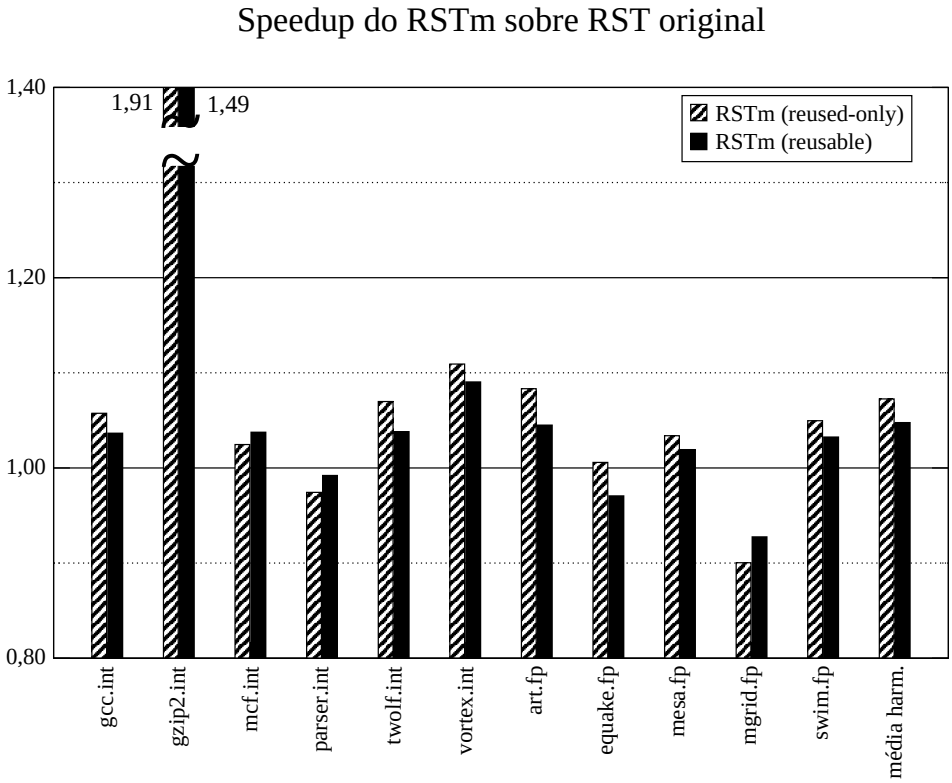

Figura 5.17: *Speedup* sobre o RST original (configuração *B* das memórias)

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Neste trabalho, foi apresentado um mecanismo de reuso especulativo de traços com instruções de acesso à memória chamado RSTm (*Reuse through Speculation on Traces with Memory*). Este mecanismo estende o RST original, proposto por Pilla (2004a), de modo que o conjunto de instruções passa a permitir que instruções de carga e escrita na memória façam parte dos traços.

Num primeiro momento, o estudo procurou determinar os limites superiores de ganho que um mecanismo de reuso especulativo com instruções de carga/escrita poderia entregar para a arquitetura. Após ficar determinado que haveria potencial para a exploração da técnica, seguiu-se a elaboração do funcionamento do mecanismo (tabelas auxiliares a serem usadas, ordem de pesquisa nas mesmas, etc.), a implementação do mecanismo voltado para processadores superescalares e a finalização deu-se através de diversas simulações executadas com o SimpleScalar.

A partir dos resultados obtidos através das diversas simulações descritas neste trabalho, pode-se constatar que a inclusão de instruções de acesso à memória ao mecanismo de reuso especulativo de traços em processadores superescalares fornece um acréscimo (média harmônica) de $4,74\%$ sobre o desempenho do mecanismo original. Tal acréscimo é observado quando o RSTm utiliza política de formação de traços original, em que as instruções candidatas a fazerem parte dos traços devem ter sido reusadas anteriormente a fim de serem incluídas nos mesmos. Ao analisarmos o mecanismo com a política especulativa, onde qualquer instrução, mesmo que não tenha sido reusada ainda, é considerada candidata a reuso, o acréscimo de desempenho é de $2,17\%$ (média harmônica). Todas as simulações aqui executadas foram obtidas para um subconjunto de *benchmarks* do SPEC2000int e SPEC2000fp.

Tais resultados comprovam o que havia sido determinado em (LAURINO et al., 2005), onde a política de formação de traços original apresenta desempenho superior ao da política especulativa. Para alguns *benchmarks* em específico esta diferença se mostrou de forma mais clara, como por exemplo, *gzip2.int* e *twolf.int*, onde a diferença de desempenho entre as políticas foi de aproximadamente 100%.

Além disso, os resultados atestam que o ganho de desempenho é obtido através da

composição de basicamente dois fatores: número de instruções reusadas e as suas respectivas latências. Mesmo apresentando $24,35\%$ menos instruções reusadas do que o RST original, o RSTm consegue ser $2,97\%$ (média harmônica) mais rápido que aquele, visto que as latências das instruções reusadas pelo mecanismo com memória compensam o menor número de instruções reusadas.

Um outro ponto interessante a ser observado é que existe um limite de ganho, de acordo com a quantidade de recursos adicionada ao mecanismo. Como pôde-se perceber, recursos em excesso (configuração *B* das tabelas) acaba gerando degradação de desempenho, pelo fato de os traços não se repetirem ao longo da execução dos programas. Com esta configuração, o RSTm apresentou uma perda de desempenho em relação ao RST de $1,41\%$ (média harmônica). O mesmo raciocínio vale para quando a quantidade de recursos é limitada: para a configuração *A* das tabelas, o ganho de desempenho do mecanismo ficou restrito devido a esta limitação. Com esta configuração, os ganhos foram de $0,90\%$ (média harmônica).

Constatou-se, também, que a maioria das instruções de acesso à memória reusadas pelo mecanismo possui seus dados em um dos três primeiros níveis de memória da arquitetura. Quando esses níveis têm sua latência aumentada (configuração *A* das memórias), os ganhos são de $7,59\%$ (média harmônica), ao passo que quando a memória principal tem sua latência aumentada (configuração *B* das memórias), os ganhos ficam em torno de $5,75\%$ apenas.

6.1 Contribuições

A principal contribuição desta Dissertação, com o desenvolvimento do RSTm, foi no sentido de compreender, medir e determinar que mecanismos de reuso especulativo são factíveis de serem utilizados quando combinados a instruções de carga/escrita na memória, especialmente quando implementados sobre arquiteturas superescalares.

Além disso, provou-se que a política de formação de traços é muito importante neste tipo de mecanismo e que esta afeta diretamente o desempenho final do mesmo. Desempenho este que é alcançado através de um balanceamento entre quantidade de recursos disponível ao mecanismo, de forma que os traços gerados tenham o tamanho ideal e que sejam compostos por instruções cujas latências são altas.

6.2 Trabalhos futuros

Existem algumas possibilidades de pesquisa e investigação adicional sobre as contribuições deste trabalho. Há uma versão do RST que implementa tabelas de reuso unificadas. Poderiam ser realizados estudos no sentido de analisar o desempenho desta versão com a adição de suporte ao reuso de memória.

Um outro ponto seria o estudo a respeito das instruções predominantes em cada *benchmark*, de modo que fosse determinado o perfil de cada um deles e de como o desempenho geral poderia ser melhorado com base nas características do conjunto de *benchmarks* analisados. Dessa forma alguns comportamentos observados no mecanismo poderiam ser melhor entendidos.

Ainda, o RSTm poderia ter suporte do compilador e de geradores de perfil, que forneceriam um código otimizado para a arquitetura do mecanismo, de forma que o reuso pudesse ser maximizado.

REFERÊNCIAS

BODIK, R.; GUPTA, R.; SOFFA, M. L. Load-Reuse Analysis: design and evaluation. In: SIGPLAN CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION, 1999. **Proceedings...** New York: ACM, 1999. p.64–76.

BURGER, D.; AUSTIN, T. M. The SimpleScalar tool set, version 2.0. **SIGARCH Comput. Archit. News**, New York, NY, USA, v.25, n.3, p.13–25, 1997.

COSTA, A. T. da. **Exploiting Dynamically the Reuse of Traces in Processor Architecture Level**. 2001. PhD Thesis — COPPE-UFRJ.

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. The Dynamic Trace Memoization Reuse Technique. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURE AND COMPILER TECHNIQUES, 9., 2000, Philadelphia, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.92–99.

COSTA, A. T. da; FRANÇA, F. M. G.; CHAVES FILHO, E. M. Exploiting Reuse with Dynamic Trace Memoization: evaluating architectural issues. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 12., 2000, São Pedro. **Proceedings...** São Paulo: SBC, 2000a. p.163–172.

GABBAY, F.; MENDELSON, A. **Speculative Execution based on Value Prediction**. Israel: Technion–Israel Institute of Technology, 1996. (Technical Report. EE Dept. #1080).

GABBAY, F.; MENDELSON, A. Using Value Prediction to Increase the Power of Speculative Execution Hardware. **ACM Transactions on Computer Systems**, New York, v.16, n.3, p.234–270, 1998.

GONZÁLEZ, A.; TUBELLA, J.; MOLINA, C. Trace-Level Reuse. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING, ICPP, 1999, Wakamatsu, Japan. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.30.

HEIL, T. H.; SMITH, Z.; SMITH, J. E. Improving branch predictors by correlating on data values. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITEC-

TURE, MICRO, 32., 1999, Haifa, Israel. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1999. p.28.

HENNESSY, J. L.; PATTERSON, D. A. **Computer Architecture**: a quantitative approach. 3rd ed. San Francisco: Morgan Kaufmann, 2003.

HENNING, J. L. SPEC CPU2000: measuring cpu performance in the new millennium. **Computer**, Los Alamitos, CA, USA, v.33, n.7, p.28–35, 2000.

HUANG, J.; LILJA, D. J. Exploiting basic block value locality with block reuse. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, HPCA, 5., 1999, Orlando, FL, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.106–114.

HUANG, J.; LILJA, D. J. Exploring Sub-Block Value Reuse for Superscalar Processors. In: INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, PACT, 2000., 2000, Philadelphia, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.100.

HWANG, K. **Advanced Computer Architecture**: parallelism, scalability, programmability. [S.l.]: McGraw-Hill Higher Education, 1992. 771p.

INTEL. **Itanium**^{tm} **Processor Microarchitecture Reference**. [S.l.], 2000.

JACOBSEN, E.; ROTENBERG, E.; SMITH, J. E. Assigning Confidence to Conditional Branch Predictions. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 29., 1996, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p.142–152.

JIN, L.; CHO, S. A Characterization Study on Memory Value Reuse. In: WORKSHOP ON MEMORY PERFORMANCE ISSUES, WMPI; INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE COMPUTER ARCHITECTURE, HPCA, 2006, Austin, TX, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006.

JOHNSON, M. **Superscalar Microprocessor Design**. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

LAURINO, L. S.; PILLA, M. L.; SANTOS, T. S. G. dos; NAVAUX, P. O. A. Reuso de Traços com Loads em Arquiteturas Superescalares. In: WORKSHOP EM SISTEMAS COMPUTACIONAIS DE ALTO DESEMPENHO, WSCAD, 6., 2005, Rio de Janeiro, RJ. **Anais...** Rio de Janeiro: SBC, 2005. p.49–56.

LEE, J. K. F.; SMITH, A. J. Branch Prediction Strategies and Branch Target Buffer Design. **Computer**, Los Alamitos, CA, USA, v.17, n.1, p.6–22, Jan. 1984.

LIPASTI, M. **Value Locality and Speculative Execution**. 1997. PhD Thesis — Carnegie Mellon University.

LIPASTI, M. H.; SHEN, J. P. Exceeding the dataflow limit via value prediction. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 29., 1996, Paris, France. **Proceedings...** Los Alamitos: IEEE Computer Society, 1996. p.226–237.

LIPASTI, M. H.; WILKERSON, C. B.; SHEN, J. P. Value Locality and Load Value Prediction. **ACM SIGPLAN Notices**, New York, v.31, n.9, p.138–147, 1996.

MCFARLING, S. **Combining Branch Predictors**. [S.l.: s.n.], 1993. (TN-36).

ONDER, S.; GUPTA, R. Load and Store Reuse using Register File Contents. In: INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, ICS, 15., 2001, Sorrento, Italy. **Proceedings...** New York: ACM Press, 2001. p.289–302.

PARCERISA, J.-M.; GONZÁLEZ, A. Reducing Wire Delay Penalty through Value Prediction. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 33., 2000, Monterey, CA, USA. **Proceedings...** New York: ACM Press, 2000. p.317–326.

PILLA, M. L. **RST**: reuse through speculation on traces. 2004a. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

PILLA, M. L.; CHILDERS, B. R.; COSTA, A. T. da; FRANÇA, F. M. G.; NAVAUX, P. O. A. A Speculative Trace Reuse Architecture with Reduced Hardware Requirements. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 18., 2006, Ouro Preto. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006. p.47–54.

PILLA, M. L.; NAVAUX, P. O. A.; CHILDERS, B. R.; COSTA, A. T. da; FRANÇA, F. M. G. Value predictors for reuse through speculation on traces. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, 16., 2004, Foz do Iguaçu. **Proceedings...** Washington: IEEE Computer Society, 2004. p.48–55.

PILLA, M. L.; NAVAUX, P. O. A.; COSTA, A. T. da; FRANÇA, F. M. G.; CHILDERS, B. R.; SOFFA, M. L. The Limits of Speculative Trace Reuse on Deeply Pipelined Processors. In: SYMPOSIUM ON COMPUTER ARCHITECTURE AND HIGH PERFORMANCE COMPUTING, SBAC-PAD, 15., 2003, São Paulo. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003. p.36–44.

REINMAN, G.; CALDER, B. Predictive techniques for aggressive load speculation. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 31., 1998, Dallas, TX, USA. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1998. p.127–137.

ROSE, C. F. D. D.; NAVAUX, P. O. A. **Arquiteturas Paralelas**. Porto Alegre: Sagra Luzzatto, 2003. 152p.

SANTOS, R. R. dos. **DCE**: the Dynamic Conditional Execution in a multipath control independent architecture. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SATHE, R.; WANG, K.; FRANKLIN, M. Techniques for Performing Highly Accurate Data Value Prediction. **Microprocessors and Microsystems**, [S.l.], v.22, n.6, p.303–313, 1998.

SAZEIDES, Y.; SMITH, J. E. The Predictability of Data Values. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.248–258.

SHARANGPANI, H.; ARORA, K. Itanium Processor Microarchitecture. **IEEE Micro**, Los Alamitos, CA, USA, v.20, n.5, p.24–43, 2000.

SODANI, A. **Dynamic Instruction Reuse**. 2000. PhD Thesis — University of Wisconsin-Madison.

SODANI, A.; SOHI, G. S. Dynamic instruction reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, ISCA, 24., 1997. **Proceedings...** New York: ACM Press, 1997. p.194–205.

SODANI, A.; SOHI, G. S. An Empirical Analysis of Instruction Repetition. In: INTERNATIONAL CONFERENCE ON ARCHITECTURAL SUPPORT FOR PROGRAMMING LANGUAGES AND OPERATING SYSTEMS, ASPLOS, 8., 1998, San Jose, CA, USA. **Proceedings...** New York: ACM Press, 1998. p.35–45.

SODANI, A.; SOHI, G. S. Understanding the Difference Between Value Prediction and Instruction Reuse. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 31., 1998. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998a. p.205–215.

STALLINGS, W. **Arquitetura e Organização de Computadores**. 5.ed. São Paulo: Prentice Hall, 2002. 786p.

VIANA, L. M. F. A. **Memorização Dinâmica de Traces com Reuso de Valores de Instruções de Acesso à Memória**. 2002. Dissertação (Mestrado em Ciência da Computação) — COPPE–UFRJ, Rio de Janeiro.

WANG, K.; FRANKLIN, M. Highly accurate data value prediction using hybrid predictors. In: ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, MICRO, 30., 1997. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.281–290.

YEH, T.-Y.; PATT, Y. N. Two-level adaptive training branch prediction. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991. **Proceedings...** New York: ACM Press, 1991. p.51–61.

YEH, T. Y.; PATT, Y. N. Alternative implementations of two-level adaptive branch prediction. In: INTERNATIONAL SYMPOSIUM ON COMPUTER ARCHITECTURE, 19., 1992, Gold Coast, Australia. **Proceedings...** Los Alamitos: IEEE Computer Society, 1992. p.124–134.