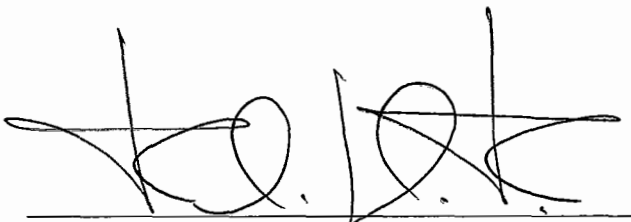
O IMPACTO DO REUSO DE TRAÇOS INTERNOS A LAÇOS

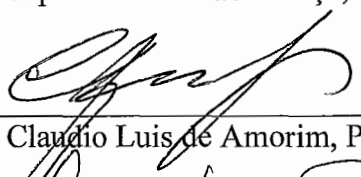
Andrey Matheus Coppieters

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

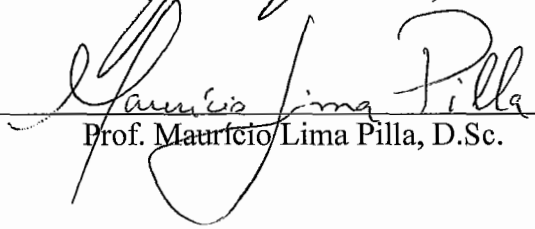
Aprovada por:



Prof. Felipe Maia Galvão França, Ph.D.



Prof. Claudio Luis de Amorim, Ph.D.



Prof. Maurício Lima Pilla, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2008

COPPIETERS, ANDREY MATHEUS

O Impacto do Reuso de Traços Internos a Laços [Rio de Janeiro] 2008

XI, 88 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2008)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Reuso Dinâmico de Traços

2. Arquitetura de Processador

I. COPPE/UFRJ II. Título (série)

Agradecimentos

A Deus, por sempre me guiar no meu caminho.

Aos meus pais, Adriano e Angela, a quem devo o suporte e amor de todos estes anos e que sempre me ensinaram a não desistir dos meus objetivos.

Aos Professores Felipe França e Valmir Barbosa, por terem acreditado no meu potencial e por terem me encorajado no momento mais difícil e crucial do meu curso.

Ao Professor Amarildo da Costa, pela ajuda e orientações nos primeiros passos deste trabalho.

Em especial, ao Professor Maurício Lima Pilla, por sua amizade, gentileza e presteza em compartilhar comigo seu conhecimento, me proporcionando condições de desenvolver este trabalho.

A Paulo Lipke e Leandro Marzulo, pela amizade e o apoio constante.

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

O IMPACTO DO REUSO DE TRAÇOS INTERNOS A LAÇOS

Andrey Matheus Coppieters

Março/2008

Orientador: Felipe Maia Galvão França

Programa: Engenharia de Sistemas da Computação (PESC)

Este trabalho identifica as estruturas de laço dos códigos dos programas como um padrão para a detecção de seqüências de instruções redundantes. Adotando esta estratégia, adaptou-se uma arquitetura superescalar com um mecanismo de reuso especulativo localizado, capturando instruções e *traces* reusáveis apenas internamente aos laços. Observou-se o mesmo nível de desempenho e uma redução suave no número de acessos às estruturas de memória (em até 12% para *Memo_Table_T* e em média 0,63% para todas as estruturas) quando comparado ao reuso empregado por todo o código. Esta manutenção no número total de acessos às estruturas foi decorrente da compensação observada entre a redução nos acessos à Tabela de Reuso e o acréscimo dos acessos às *Caches L1* e à Tabela do Preditor de Desvios.

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

THE IMPACT OF TRACE REUSE INSIDE LOOPS

Andrey Matheus Coppieters

March/2008

Advisor: Felipe Maia Galvão França

Department: Computing Systems Engineering

This work points out that loop structures inside program codes is the main pattern to prefetch as reusable instruction sequences. Adopting this strategy, some adaptations have been made to a speculative reuse mechanism of a superscalar architecture, so it would be able to dynamically detect the beginning and ending of loops as boundaries in order to keep the reuse mechanism turned on only inside loops. It was observed that the performance level was mantained and also a slight reduction on the number of memory structure accesses (until 12% to *Memo_Table*_T and a 0.63% average to all structures) when compared to the reuse mechanism applied to the entire code. Regarding the maintenance of the total number of memory accesses, this could be explained by a balance between the reduction of accesses to the unified Reuse Table and the increase of accesses to the first level *caches*, so as to the Branch Predictor Table.

CONTEÚDO

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1 – Introdução

O reuso dinâmico de instruções redundantes tem se apresentado como uma das mais promissoras vertentes para obtenção de um aumento considerável de desempenho na arquitetura de computadores, sob a necessidade da implementação de pouco hardware adicional quando comparada a outras técnicas propostas. A técnica de reuso consiste em identificar, em tempo de execução, instruções que pertençam a um domínio pré-estabelecido, armazená-las em uma estrutura conhecida como tabela de reuso e detectar novas ocorrências daquelas mesmas instruções que possuam valores de entradas reincidentes. O objetivo é permitir que os resultados possam ser consultados diretamente a partir da tabela de reuso e eliminar, portanto, a necessidade de uma reexecução daquelas instruções pelo caminho de dados da arquitetura.

Sob a perspectiva de ganhos promissores, a redundância nos códigos dos programas passou, em seguida, a ser explorada a nível de *traces*, poupando-se em um único ciclo de clock a reexecução de uma seqüência completa de instruções identificadas como redundantes. O mecanismo *DTM (Dynamic Trace Memoization)* apresentado por DA COSTA (2001) baseia-se nesta estratégia e apresentou ganhos na ordem de 8,5% sobre uma arquitetura-base sem o emprego do reuso.

Apesar dos bons resultados apresentados pelo *DTM*, notou-se ainda que parte considerável do código redundante não era reusado devido a indisponibilidade de alguns valores de entrada, ainda sendo calculados ou acessados da memória, no momento do teste de reuso. Para capturar estas oportunidades de reuso, que estavam sendo desperdiçadas, PILLA (2001, 2002, 2003a, 2003b) introduziu o mecanismo *RST* (*Reuse Through Speculation on Traces*), que combina as principais vantagens das técnicas de reuso e de previsão de valores, visando impulsionar ainda mais o reuso de *traces* e esconder relações de dependências de dados verdadeiras entre instruções. Uma

instrução reusável isolada, ou de entrada de um *trace,* terá especulados os valores dos seus operandos de entrada que ainda não se encontrem prontos na etapa do teste de reuso. A técnica de especulação depende diretamente de mecanismos de confiança e de recuperação, este em caso de falhas de previsão, bastante eficientes. O emprego do *RST* proporcionou um ganho médio de 7% sobre uma arquitetura com o mecanismo *DTM.*

Neste trabalho, identifica-se as estruturas de laço como um padrão nos *paths* que antecedem as seqüências de instruções redundantes. Adaptou-se uma arquitetura superescalar para uso de um mecanismo de reuso especulativo localizado, capaz de detectar dinamicamente estas regiões. Os resultados experimentais mostram a variação de *speedup* para várias configurações desta arquitetura e propõem uma análise crítica sobre os *tradeoffs* do investimento na tabela de reuso e nas estruturas de memória de primeiro nível para aumento do desempenho.

Apresenta-se uma redução do número de *traces* capturados na versão adaptada para o reuso interno a laços, comparada ao RST original, entre 1 e 12%, dependendo do benchmark, mantendo-se o mesmo nível de desempenho, além de uma redução, na média em torno de 0,63%, no número total de acessos feitos às estruturas de memória de primeiro nível. Considerando-se a possibilidade de um aumento na ordem de 100% no budget de memória, obteve-se um melhor desempenho (speedup de 1,7% contra 0,2%) aumentando-se o tamanho da tabela de reuso, ao invés de aumentar-se o tamanho das caches L1, sob pouca penalidade, em torno de 0,88%, quanto ao acréscimo no número de acessos às estruturas.

Este trabalho está estruturado em 5 capítulos, incluindo esta primeira parte introdutória. No Capítulo 2 é apresentado o estado da arte, referenciando trabalhos anteriores relacionados a exploração da técnica de reuso objetivando o aumento de

desempenho da arquitetura. No Capítulo 3, apresenta-se o estudo e o *profiling* realizados sobre os trechos de código dos programas reusados pelo *RST* , apontando as estruturas de laços como foco para exploração localizada do reuso e obtenção de novos resultados. No Capítulo 4 são apresentados o ambiente experimental, a comparação entre os resultados obtidos pela execução dos programas de teste em diferentes versões e configurações do simulador. No Capítulo 5 são apresentadas as conclusões desta dissertação.

CAPÍTULO 2 – Trabalhos Relacionados

Neste capítulo serão apresentados os conceitos e resultados, provenientes de trabalhos anteriores, na área de exploração de computações redundantes no nível de *traces*, que compõem a base fundamental desta Dissertação.

2.1 Reuso Dinâmico de Instruções

Dentre os principais fatores que determinam o tempo de execução de um programa, muitos esforços têm sido direcionados à redução das instruções a serem executadas dinâmicamente pela arquitetura. Com base no fato de que a codificação é feita de forma concisa, objetivando ser a mais genérica possível e capaz de processar um conjunto de dados de entrada, o resultado observado é a recorrência freqüente a valores de entrada já utilizados, que reforça o perfil da localidade temporal e espacial na execução dos programas (HENESSY; PATTERSON, 2003).

Sob este prisma, a redução do número de instruções a serem executadas dinâmicamente pode ser obtida pelo emprego da técnica de reuso dinâmico de instruções (SODANI; SOHI, 1997). Sua implementação consiste no armazenamento temporário dos resultados de instruções, para o uso posterior por instâncias futuras destas, que acessariam os mesmos operandos e produziriam os mesmos valores repetidamente, sem a necessidade de reexecução. As instruções que apresentam este comportamento são referenciadas neste trabalho como redundantes.

Poupando-se a passagem destas instruções pelos estágios de execução, há uma redução significativa dos conflitos estruturais a medida que a demanda pelos recursos da arquitetura é reduzida, e dos caminhos críticos de execução, uma vez que o resultado daquela instrução reusada passa a estar disponível com maior antecedência.

Este benefício ainda possibilita a geração dos resultados de uma cadeia inteira de instruções dependentes em um único ciclo de clock.

O reuso de instruções redundantes pode ser feito tanto dinâmica quanto estaticamente. Porém, a opção pela primeira deve-se ao grande esforço a ser empregado no compilador para a realização do reuso de maneira estática, que envolve questões como *loop unrolling*, *in-lining* de funções, assim como registradores suficientes para alocação de variáveis decorrentes destas optimizações.

Dentre as pioneiras iniciativas visando à implementação em hardware do conceito de reuso dinâmico de instruções, foram apresentados quatro esquemas que utilizam uma estrutura para o armazenamento dos resultados das instruções chamada *Reuse Buffer (RB)* (SODANI; SOHI, 1997; SODANI, 2000). O teste de reuso é feito sobre uma pesquisa no *RB*, indexada pelo valor do *Program Counter (PC)* da instrução.

Os esquemas se diferenciam quanto à forma como a qual as instruções são identificadas como reusáveis, o que determina quais informações são armazenadas no *RB*, como o teste de reuso é realizado e como as informações no *RB* são atualizadas, como ilustrado pela Figura 2.1 (COSTA, 2001).

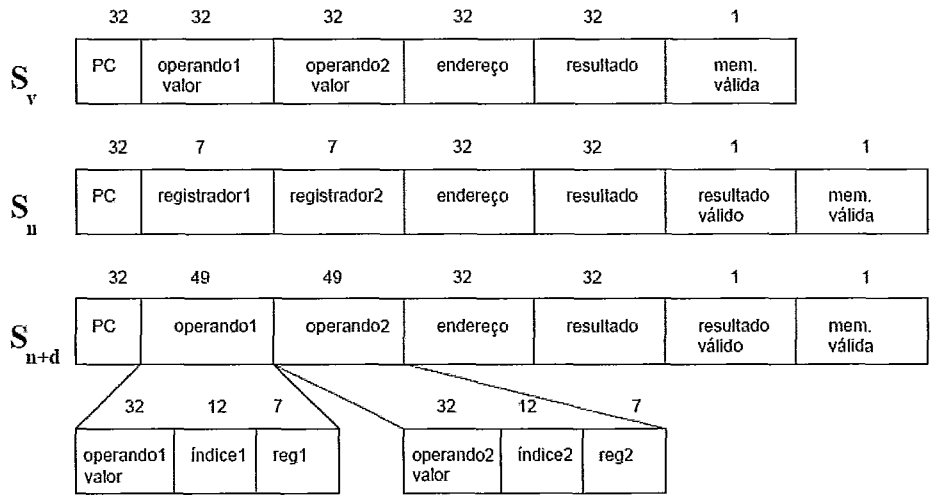


Figura 2.1 – Entradas em RB para os esquemas S_v, S_n e S_{n+d}

O esquema S_v checa os valores dos operandos de cada instrução, enquanto que o S_n checa pelos nomes dos operandos (identificadores dos registradores). Os esquemas S_{v+d} e S_{n+d} estendem aqueles checando por cadeias de instruções dependentes (SODANI; SOHI, 1997; SODANI, 2000). Exemplificando, as Figuras 2.2 e 2.3 (COSTA, 2001). mostram como é feito o armazenamento em *RB* para os esquemas S_n e S_{v+d}.

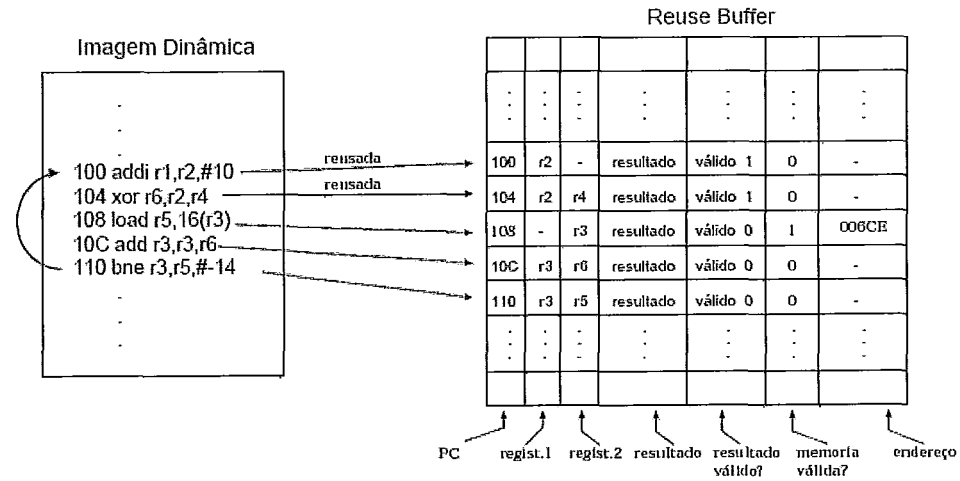


Figura 2.2 - Reuso no Esquema Sn

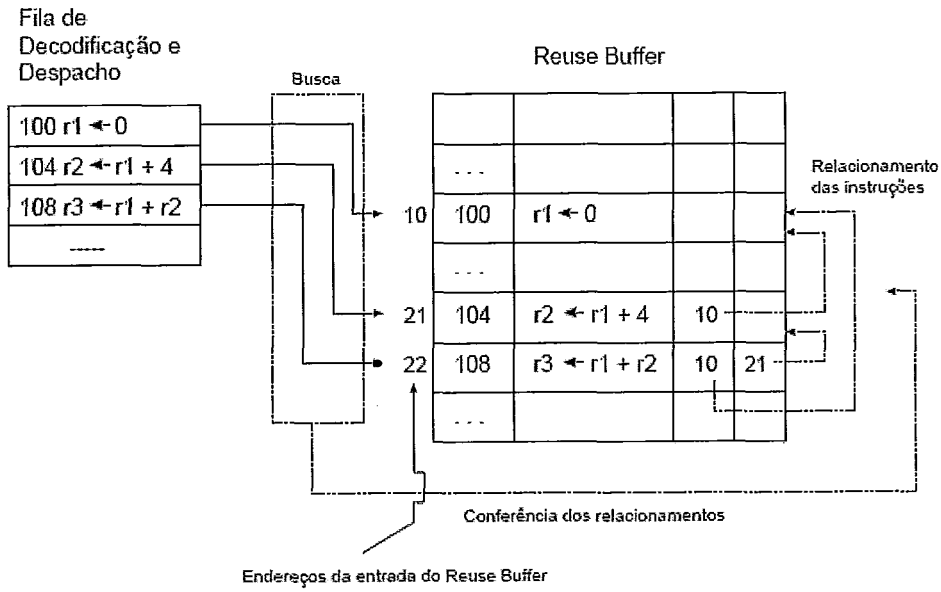


Figura 2.3 - Reuso no Esquema Sv+d

Nesta abordagem inicial, as instruções redundantes foram agrupadas em dois casos: *squash reuse* e *general reuse* (SODANI; SOHI, 1997). O *squash reuse* refere-se ao reuso de instruções que foram executadas após uma previsão incorreta de um desvio, mas que podem ter seus resultados eventualmente reaproveitados, enquanto que o *general reuse* refere-se a outras situações possíveis.

2.2 Previsão de Valores

Uma técnica que também explora a redundância computacional, porém de forma especulativa, é a de Previsão de Valores. Tenta-se aumentar o desempenho da arquitetura ao viabilizar a execução de instruções cujos operandos de entrada ainda não se encontram disponíveis, prevendo os valores destes com base em ocorrências anteriores. Esta técnica é amplamente conhecida pela sua aplicação na previsão de desvios e pela dependência de implementações de mecanismos de confiança e de recuperação eficientes (SODANI; SOHI, 1998).

Tanto o reuso de instruções dinâmicamente quanto a previsão de valores resolve muitos dos conflitos de dependência verdadeira de dados na execução de um programa. A diferença entre estas técnicas está na especulação feita pela Previsão de Valores, que resulta na variação entre o quanto de computação redundante cada técnica é capaz de detectar. O reuso de instruções verifica os resultados antes de usá-los (*early validation*), enquanto que a previsão de valores utiliza os resultados de maneira especulativa e faz a validação posteriormente (*late validation*) (SODANI; SOHI, 1998).

Quando há a previsão incorreta de um valor, todas as instruções dependentes do mesmo são reexecutadas. Esta execução é atrasada pela latência da validação do estágio de verificação do mecanismo de previsão de valores (SODANI; SOHI, 1998).

Quanto à resolução de desvios, tanto o reuso de instruções quanto a previsão de valores é capaz de fazê-la com maior antecedência, ao identificar a redundância nas cadeias de instruções dependentes que levam à instrução de desvio, reduzindo assim a penalidade decorrente de um desvio mal previsto. Enquanto o reuso de instruções reduz o peso desta penalidade pelo fato de realizar a sua validação no estágio de decodificação, a técnica de previsão de valores pode impactar de forma contrária devido a gerar um maior número de desvios mal previstos e ainda atrasar a resolução do desvio (SODANI; SOHI, 1998).

O reuso de instruções tende a reduzir a requisição pelos recursos (unidades funcionais) da arquitetura, uma vez que as instruções reusadas não são executadas. A técnica de previsão de valores, entretanto, pode aumentar esta demanda devido a necessidade de reexecução de instruções previamente executadas com valores erroneamente previstos (SODANI; SOHI, 1998).

A combinação de ambas as técnicas, pela execução especulativa baseada na previsão de valores e no reuso de valores, oferece todas as suas vantagens para o aumento do desempenho na execução de programas. Uma das propostas de implementação funcionaria de forma que quando um operando não estivesse presente no *Reuse Buffer*, mas o mesmo fosse encontrado na tabela de previsão de valores, assume-se que o teste de reuso foi realizado com sucesso e os resultados armazenados no *RB* devem ser encaminhados (LIAO; SHIEH, 2002).

2.3 Reuso de Blocos Básicos

Um segundo passo, após os bons resultados obtidos com o reuso isolado de instruções dinâmicamente (SODANI; SOHI, 1997), foi a introdução de um mecanismo de reuso no nível de blocos básicos, tomando como base a alta correlação entre as entradas e saídas

de cadeias de instruções dependentes (HUANG; LILJA, 1999). Portanto, um grupo de instruções passa a ser reusado dinâmicamente, e não apenas uma instrução isoladamente.

Nesta proposta, cada instrução do programa é pertencente a um bloco básico e cada um destes é identificado dinâmicamente, ou estaticamente caso o compilador tenha feito uma marcação prévia. As instruções de desvio, incluindo chamadas e retornos de sub-rotinas, desempenham o papel de limítrofes para identificação dos blocos básicos, os quais têm seus contextos de entrada e saída armazenados em uma estrutura chamada *Block History Buffer (BHB)* (HUANG; LILJA, 1999). A Figura 2.4 (COSTA, 2001) ilustra a composição de uma entrada em *BHB*.

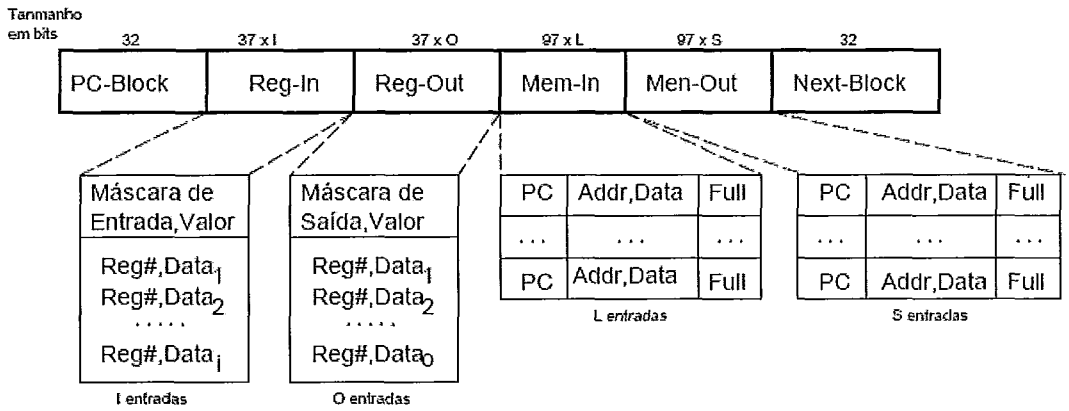


Figura 2.4 – Formato de uma entrada no Block History Buffer

As instruções de acesso à memória não são reusadas, na maioria dos mecanismos, devido aos possíveis efeitos colaterais levantados pela questão do *memory aliasing*, a qual descreve a alta complexidade de sabermos se outra instrução de acesso à memória precedente alterou o valor armazenado no endereço referenciado pela instrução.

A Figura 2.5 (HUANG; LILJA, 1999) ilustra a implementação de um processador contando com o mecanismo para reuso de blocos básicos.

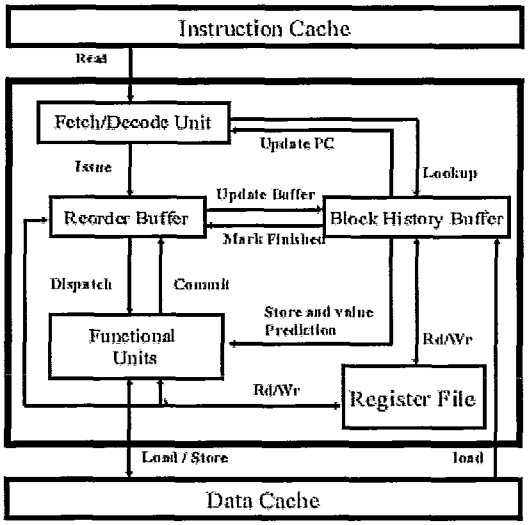


Figura 2.5 – Modelo de processador empregando o reuso no nível de blocos básicos

2.4 Reuso de Sub-blocos

Conjuntos maiores de instruções geralmente apresentam menor incidência de oportunidades de reuso do que os grupos menores, porém obviamente o ganho gerado é maior quando são reusados. É fundamental encontrar um ponto de equilíbrio de forma a se tirar proveito do maior aumento de desempenho possível.

Usando uma estrutura fixa como o *BHB*, para o armazenamento das informações dos blocos básicos reusáveis, não garante a retenção de todos os blocos identificados já que o tamanho de alguns pode ultrapassar o limite da entrada no *BHB*. Aliás, mostrou-se que até 25% dos blocos básicos deixavam de ser retidos no *BHB* devido seu tamanho gerar uma quantidade de informação que excedia a capacidade de armazenamento da entrada desta estrutura (HUANG; LILJA, 1999).

Uma nova proposta para contornar estas questões foi dividir os blocos básicos, cujas fronteiras são as instruções de desvio, em unidades menores, sub-blocos, para a aplicação da técnica de reuso de valor. Esta sub-divisão pode ser feita dinâmicamente, utilizando o número de instruções, de entradas e saídas, ou a chegada à uma instrução de store para determinar as fronteiras do sub-bloco; ou sob suporte do compilador, que considera o fluxo de dados para balancear a granularidade do reuso com o freqüência de oportunidades de reuso (HUANG, LILJA, 2000).

Os resultados do emprego desta técnica mostraram que os sub-blocos são melhores candidatos ao reuso de valores do que os blocos básicos.

2.5 Reuso de *Traces*

O reuso de seqüencias de instruções (*traces*) dinâmicamente foi o passo seguinte na técnica de reuso de valor, sendo independente do auxílio do compilador e abrangendo um domínio maior de instruções reusáveis em um único conjunto (GONZÁLEZ; TUBELLA; MOLINA, 1998; COSTA; FRANÇA, 1999). As instruções de desvio podem fazer parte de um trace, cujas instruções limítrofes costumam ser as de acesso à memória, ou de operação sob ponto flutuante ou de chamada ao sistema operacional (requer a mudança de *status* do processador, de modo usuário para modo protegido), conforme ilustrado pela Figura 2.6 (COSTA, 2001).

De modo similar ao mecanismo de reuso de blocos básicos, os contextos de entrada e saída do trace, ou seja, os registradores lidos e escritos pelo mesmo, são armazenados com os seus valores em uma estrutura, indexada pelo valor do *PC* da instrução inicial do trace. O contexto de entrada contém os operandos fonte e seus respectivos valores, referenciados por instruções que compõem o trace mas que são produzidos por instruções fora do mesmo. O contexto de saída é composto pelos

operandos destino e seus valores, referenciados e produzidos pela execução das instruções que compõem o trace.

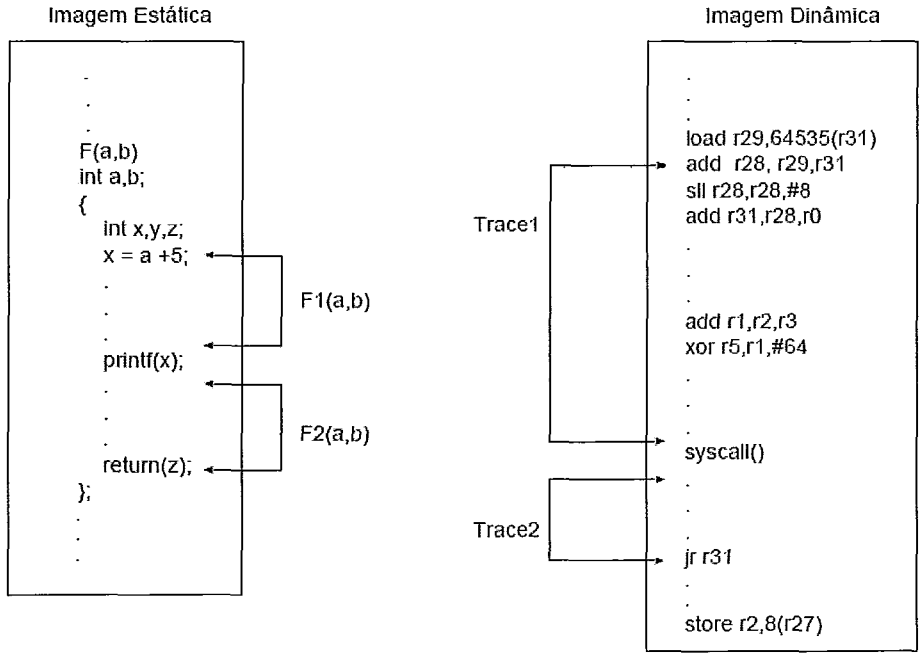


Figura 2.6 – Mapeamento de traces sob a limitação de instruções com efeitos colaterais

Um trace é redundante, como o da Figura 2.7 (COSTA, 2001) quando seu contexto de entrada previamente armazenado é idêntico ao contexto da arquitetura no momento da busca da instrução inicial do trace. Quando há a identificação de uma oportunidade de reuso do trace, os valores armazenados na estrutura são escritos nos registradores da arquitetura que compunham o contexto de saída, assim poupando as instruções integrantes do trace da passagem pelos estágios de execução do pipeline. O reuso na granularidade de *traces* significa reusar todas as instruções pertencentes ao trace identificado como redundante de uma única vez (COSTA, 2001).

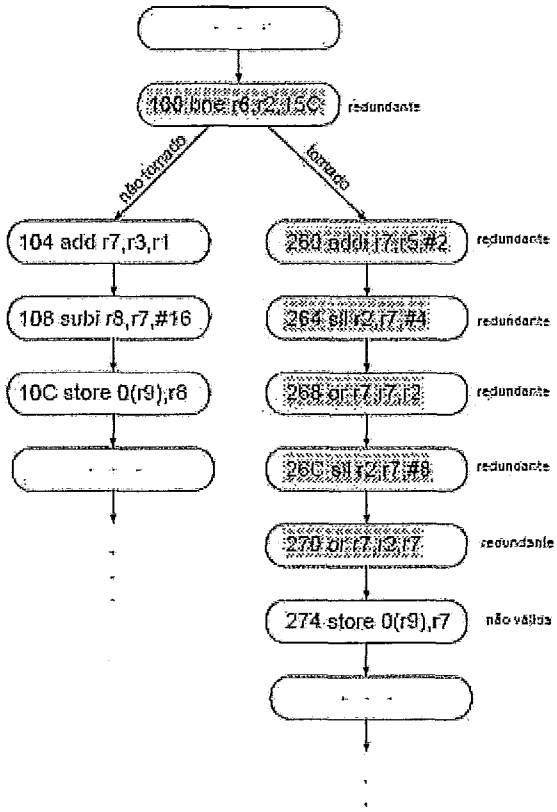


Figura 2.7 – Exemplo de trace redundante

2.6 Memorização Dinâmica de *Traces* (DTM)

Inspirado no modelo *Function Memoization* (MICHIE, 1968), uma técnica de software para eliminação de redundância em tempo de execução, foi desenvolvido um mecanismo similar em hardware, o *DTM* (*Dynamic Trace Memoization*), que emprega tabelas de memorização para permitir que funções, que não tenham efeitos colaterais, retornem imediatamente valores já conhecidos de execuções prévias com base nos mesmos argumentos de entrada (COSTA; FRANÇA; *et al* , 2000).

O *DTM* implementa o reuso a nível de *traces*, compostos apenas por seqüências de instruções redundantes e que pertençam ao domínio-alvo, que abrange todas as instruções exceto as de acesso à memória, de ponto flutuante e de chamada ao sistema operacional. Apesar de não marcar a instrução de acesso à memória como

redundante, o *DTM* trata esta em duas sub-instruções: uma para o cálculo do endereço e outra para o acesso propriamente dito. O cálculo do endereço associado pode ser eventualmente reusado (COSTA, 2001).

O mecanismo *DTM* é uma técnica de memorização que realiza a identificação da redundância (construção dos *traces*) dinâmicamente, o armazenamento e a identificação das oportunidades de reuso em *hardware*.

A detecção de instruções ou *traces* redundantes consiste em verificar se a instância dinâmica da instrução possui os mesmos operandos de entrada já processados por uma instância anterior. Para *traces*, esta tarefa é ainda mais complexa, pela obrigatoriedade de ser composto apenas por instruções redundantes e que não tenham efeitos colaterais.

2.6.1 A Construção de *Traces* no *DTM*

Na implementação DTM, os *traces* redundantes são construídos a partir de informações das instruções dinâmicas redundantes. Para tal, é empregada uma tabela chamada *Memo Table G*, ou Tabela de Memorização Global, que mantém apenas as informações de instâncias dinâmicas de instruções que pertençam ao domínio de instruções válidas. Cada entrada na *Memo Table G*, como visto na Figura 2.8 (COSTA, 2001), está associada a uma instância dinâmica de uma instrução, indexada pelo seu valor de *PC* (*Program Counter*). A tabela não armazena o código das instruções, e sim apenas as informações que serão úteis posteriormente para a identificação da oportunidade de reuso e a eventual efetuação do reuso, conforme exibido pela Figura 2.9 (COSTA, 2001). O campo *PC* contém o endereço da instrução na memória. Os campos S_{v1} e S_{v2} mantêm os valores dos operandos usados pela instância dinâmica da instrução. O campo *res/targ* armazena o resultado de uma instrução aritmética ou lógica, ou o endereço-alvo

de uma instrução de desvio de controle de fluxo de execução. Os bits *jmp* e *brc* identificam se a instrução é um *jump* ou um *branch*, respectivamente. O bit *btaken* indica se o desvio identificado por *brc* foi tomado ou não (COSTA; FRANÇA, *et al*, 2000; COSTA, 2001).

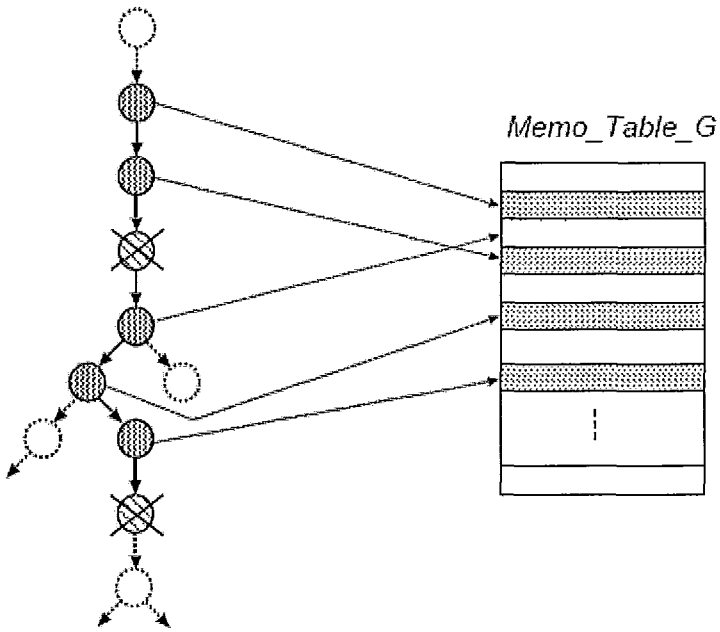


Figura 2.8 – Armazenamento de instruções dinâmicas em Memo Table G

32	32	32	32	1	1	1
pc	sv1	sv2	res/targ	jmp	brc	btaken

Figura 2.9 – Entrada em *Memo Table G*

Dinâmicamente, o *DTM* classifica as instruções como redundantes ou não. Caso a instrução não pertença ao domínio de instruções válidas, é simplesmente marcada como não redundante. Caso seja uma instrução válida, os valores atuais dos seus operandos e do seu *PC* são comparados com os armazenados na *Memo Table G*. Se o resultado desta validação for negativo a instrução é marcada como não redundante, é criada uma entrada na tabela para aquela instrução e os seus valores são armazenados

devidamente. Em caso positivo, a instrução é marcada como redundante e não é criada a entrada na *Memo Table G* (COSTA; FRANÇA, *et al*, 2000).

Diferentemente de outras implementações, o *DTM* inclui as instruções de controle no domínio de instruções passíveis de reuso, como usualmente se faz com as dos tipos aritmética e lógica. Considera-se que uma instrução de desvio condicional é redundante caso os valores dos registradores no estado arquitetural correspondem aos valores armazenados na *Memo Table G* para aquela instrução. As instruções de desvio incondicional indireto são avaliadas conforme o seu endereço de destino, que está armazenado em um registrador. Nas instruções de acesso à memória, apesar de não presentes no domínio tratado pelo *DTM*, a operação do cálculo do endereço a ser acessado é considerada como uma instrução aritmética e terá uma entrada alocada na *Memo Table G*, sendo o resultado armazenado no campo *res/targ* (COSTA; 2001).

Caso uma instrução tenha o rótulo de redundante, o *DTM* iniciará a construção de um novo trace ou acrescentará aquela instrução ao trace em formação (atualizando os mapas contextuais de entrada e saída).

A Figura 2.10 (COSTA, 2001) ilustra a construção de um trace, que se encerra com a identificação de uma instrução rotulada como não redundante. Aquela pode não pertencer ao domínio, ou não ter uma instância anterior já executada, ou os valores armazenados no contexto arquitetural diferem dos armazenados na *Memo Table G*. O encerramento do trace também pode ocorrer caso se atinja o número limite de valores possíveis nos contextos de entrada ou saída, ou o limite de instruções de desvio presentes em um mesmo trace (COSTA; FRANÇA, *et al*, 2000).

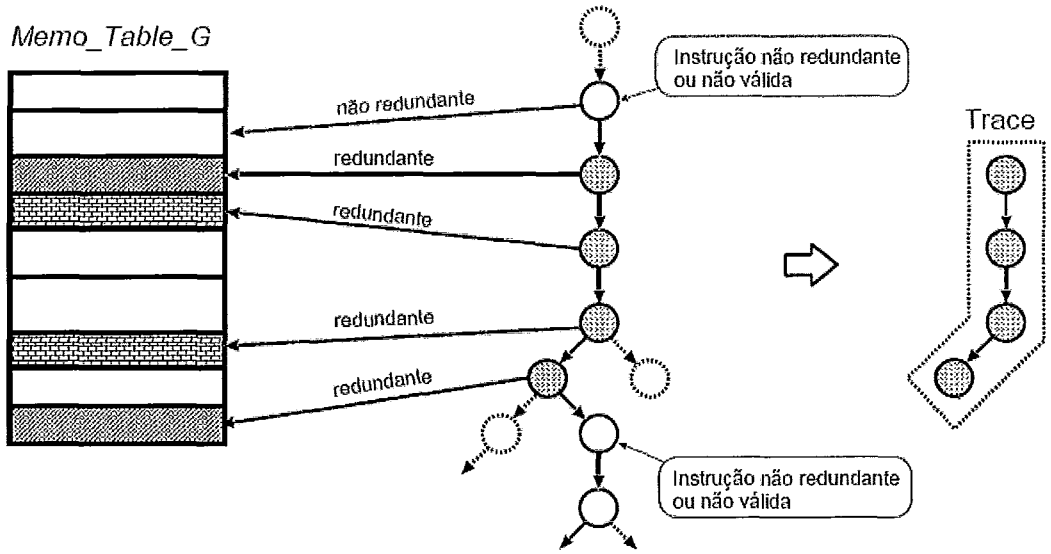


Figura 2.10 – Construção de um trace

2.6.2 A Memorização de *Traces*

O processo de construção de um trace está baseado em algumas estruturas que armazenam os seus mapas contextuais de entrada e saída, que contêm um bit para cada registrador existente na arquitetura e são atualizados a medida que cada instrução marcada como redundante é identificada. Cada bit do mapa contextual de entrada é ativado caso o registrador correspondente mantenha um valor que fôra produzido por uma instrução de fora do trace em formação. Cada bit do mapa contextual de saída é ativado caso o registrador correspondente seja o operando destino de uma instrução do trace (COSTA; FRANÇA, *et al*, 2000). A Figura 2.11 (COSTA, 2001) exemplifica este processo de construção dos mapas contextuais de um *trace*.

Conforme os bits dos mapas contextuais são ativados, o conteúdo de cada registrador, cujo bit está ativado, é acrescentado ao contexto de entrada ou saída do trace em formação. Este armazenamento é feito, a priori, em *buffers* que permitem o início da construção de um novo trace antes que o trace anterior tenha sido

completamente salvo. Ao término da construção de um trace, as informações são transferidas do *buffer* para uma tabela, totalmente associativa, que armazena os *traces* capturados, chamada *Memo Table T* ou *Tabela de Memorização de Traces*.

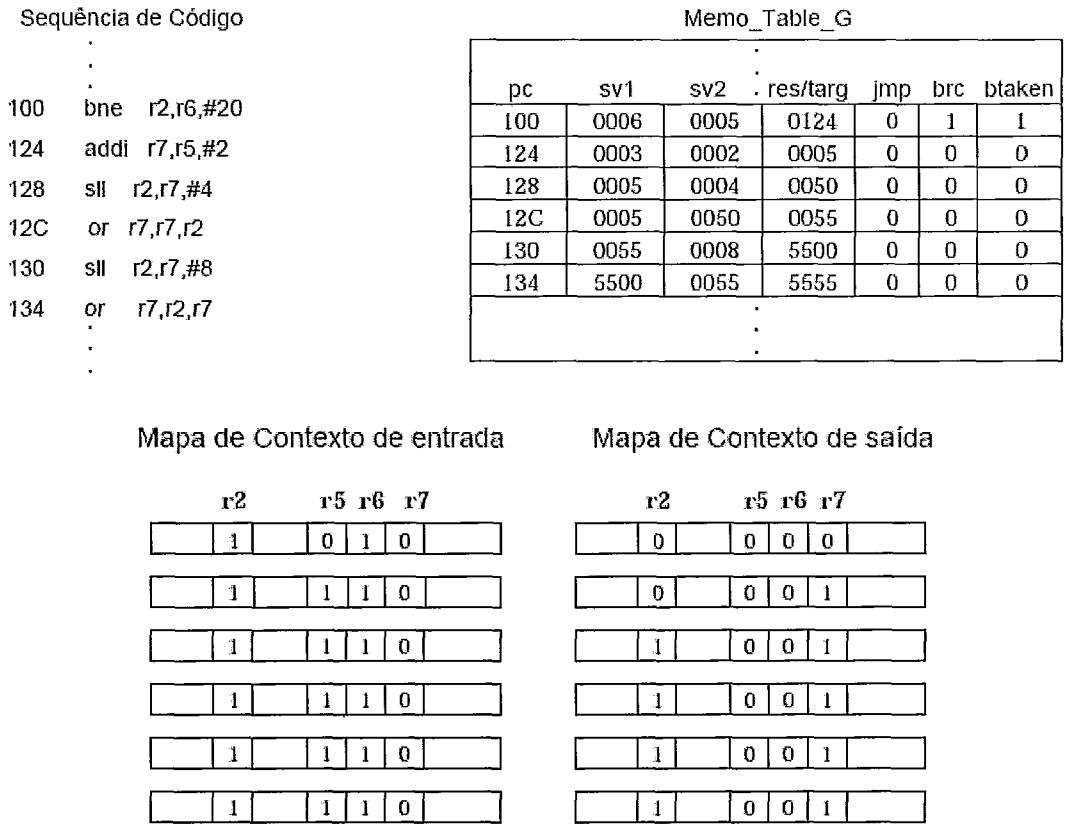


Figura 2.11 – Construção dos mapas contextuais de um trace para memorização

Cada entrada na *Memo Table T* referencia um trace construído. Todos os campos existentes no *buffer* também estão presentes nesta tabela (Figura 2.2). O campo *PC* mantém o endereço da instrução inicial do trace. O campo *NPC* armazena o endereço da próxima instrução a ser executada quando o trace for reusado, já considerando as eventuais transferências de controle de fluxo de execução (desvios condicionais ou incondicionais) que venham a existir internamente ao trace. Os campos

$icr_1,...,icr_n$ identificam os registrados que armazenam os valores do contexto de entrada do trace, os quais são armazenados nos campos $icv_1,...,icv_n$. Analogamente, os campos $ocr_1,...,ocr_n$ apontam os registradores que retêm os valores do contexto de saída do trace, enquanto que $ocv_1,...,ocv_n$ armazenam estes valores. O campo *bmask* possui uma série de bits que indicam a presença de instruções de desvio condicional no trace. O campo *btaken* armazena a informação sobre o resultado de cada instrução de desvio do trace, se foi tomado ou não. Estes dois últimos campos serão usados para alimentar o preditor de desvios (COSTA; FRANÇA, *et al*, 2000; COSTA, 2001).

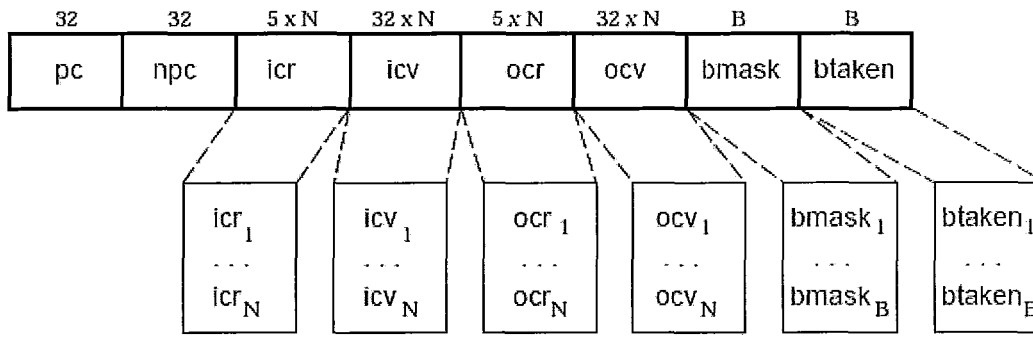


Figura 2.12 – Entrada em Memo Table T

Apesar da instrução de acesso à memória não pertencer ao domínio do *DTM* e, portanto, podendo ser responsável pelo encerramento da construção de um trace, a operação de cálculo do endereço a ser acessado por aquela instrução tem o seu resultado armazenado em um dos campos de valor do contexto de saída do trace (ocv_n). Nesta situação, o campo ocr_n correspondente será marcado como uma referência a um endereço de memória, e não a um registrador (COSTA; 2001).

Assim como a *Memo Table G*, a *Memo Table T* também não armazena os códigos das instruções que compõem o trace, mas sim apenas as informações dos seus contextos de entrada e saída, necessárias para o eventual reuso. Múltiplas instâncias de

um mesmo trace podem estar armazenadas em diferentes entradas da *Memo Table T,* pois cada instância pode ter um contexto de entrada diferente.

2.6.3 A Identificação do Reuso

Em paralelo à construção de *traces*, o *DTM* faz a verificação por instruções e *traces* que possam ser reusados, através de consulta às tabelas *Memo Table G* e a *Memo Table T.* Para cada instrução acessada na memória de instruções, é feita a comparação associativa do seu endereço com o campo *PC* de cada entrada da *Memo Table T*, de maneira a checar se aquela instrução é inicial de algum trace anteriormente construído e armazenado. Caso esta comparação retorne resultados positivos, são selecionadas todas as entradas da *Memo Table T* que satisfizeram a busca e, para cada entrada, são comparados os valores contidos nos registradores da arquitetura, que compõem o mapa de entrada do trace, com os valores armazenados nos campos icv_n daquela entrada da tabela. Havendo equivalência entre o contexto arquitetural e uma entrada da *Memo Table T*, aquele trace é identificado como redundante (COSTA; FRANÇA, *et al*, 2000; COSTA, 2001).

As pesquisas nas tabelas *Memo Table G* e *Memo Table T* são feitas concomitantemente e o *DTM* faz a escolha de quais informações reusar conforme os resultados retornados. A detecção de *traces* redundantes é análoga a de instruções, diferindo apenas quanto à estrutura consultada, já que instruções redundantes são identificadas pela *Memo Table G*, enquanto que os *traces* redundantes são detectados pela *Memo Table T.*

Na ocorrência de *misses* (retorno negativo da pesquisa) em ambas as tabelas, nada é reusado e a execução procede normalmente. Se houver *hits* (retorno de acertos da pesquisa) em ambas as tabelas, o *DTM* opta pelo reuso do trace armazenado

em *Memo Table T*, tomando como premissa o fato de que é mais válido reusar uma seqüência de instruções do que apenas uma instrução isoladamente na busca pela melhoria do desempenho. Havendo *miss* na *Memo Table G* e *hit* na *Memo Table T*, o trace redundante capturado é reusado. Caso contrário, *hit* na *Memo Table G* e *miss* na *Memo Table T*, a instrução redundante é reusada

Na detecção de um trace redundante a ser reusado, os registradores da arquitetura referenciados pelo mapa contextual de saída do trace receberão os valores de saída armazenados na *Memo Table T* para aquele trace. O registrador *PC* (*Program Counter*) receberá o valor armazenado no campo *NPC* contido na entrada do trace na tabela, e o preditor de desvios terá o seu estado atualizado pelas informações dos campos *btaken* e *bmask*. O fluxo percorrido por uma implementação que conta com um mecanismo de reuso no nível de *traces* é ilustrado na Figura 2.13 (COSTA, 2001).

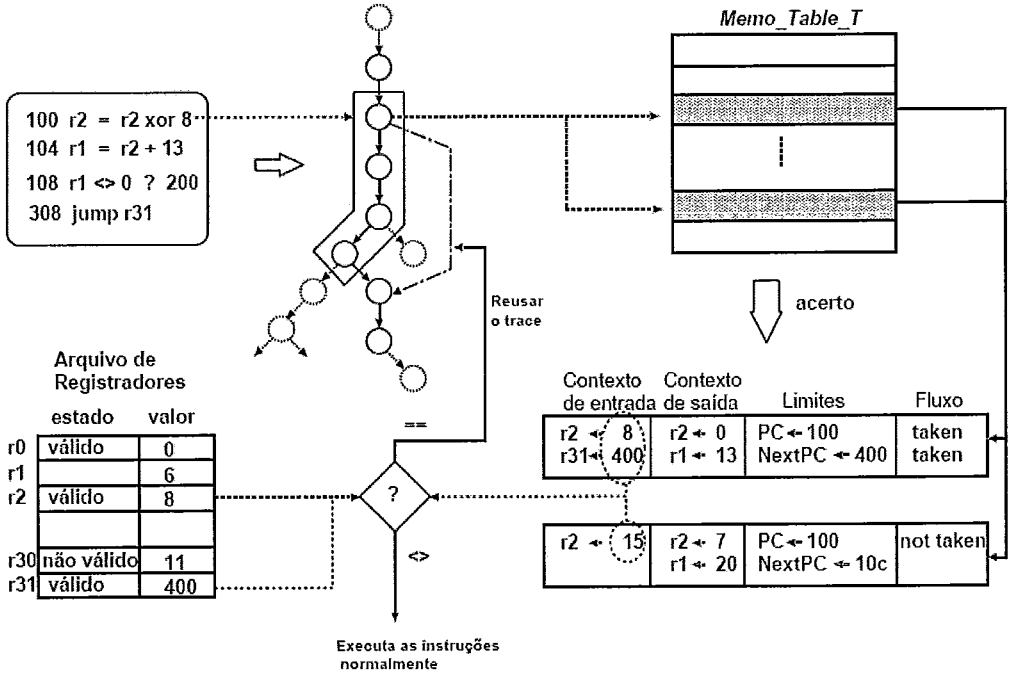


Figura 2.13 – Fluxo do processo de reuso de um trace

2.6.4 A Implementação do *DTM*

A implementação do *DTM* em uma microarquitetura é organizada em três estágios (*DS1*, *DS2* e *DS3*) que operam em paralelo aos estágios de Busca, Decodificação e Entrega (*Commit*) do pipeline, como mostrado na Figura 2.3 (COSTA; 2001)

O estágio *DS1* recebe o endereço do estágio de Busca, insere instruções na *Memo Table G* , e inicia a seleção de instruções redundantes nesta tabela, assim como a de *traces* redundantes na *Memo Table T.*

O estágio *DS2* fornece ao estágio de Decodificação o índice de cada entrada em *Memo Table G* alocada no ciclo anterior, identifica instâncias de instruções ou *traces* redundantes com base na leitura dos operandos fontes (registradores arquiteturais) feita pelo estágio de Decodificação e decide pelo reuso da instrução ou trace detectado. O estágio *DS2* requere portas de leitura do arquivo de registradores adicionais às já existentes na arquitetura convencional, para que permita que a leitura dos valores do contexto de entrada seja realizada concorrentemente com a operação regular de busca dos operandos.

O estágio *DS3* recebe do estágio de Entrega os operandos da instrução, o seu resultado e o índice de *Memo Table G* que havia sido fornecido por *DS2* ao estágio de Decodificação. Com este índice, o estágio *DS3* preenche os campos dos operandos de entrada e de resultado na *Memo Table G.* De acordo com a marcação da instrução (redundante ou não), feita pelo estágio *DS2*, o estágio *DS3* iniciará a construção de um novo trace, ou atualizará o mapa contextual de bits do trace em formação, ou finalizará a construção de um trace. Este estágio também preencherá os campos de referência aos registradores que compõem os mapas de contexto de entrada e saída no buffer temporário com base nos dados recebidos pelo estágio de Entrega.

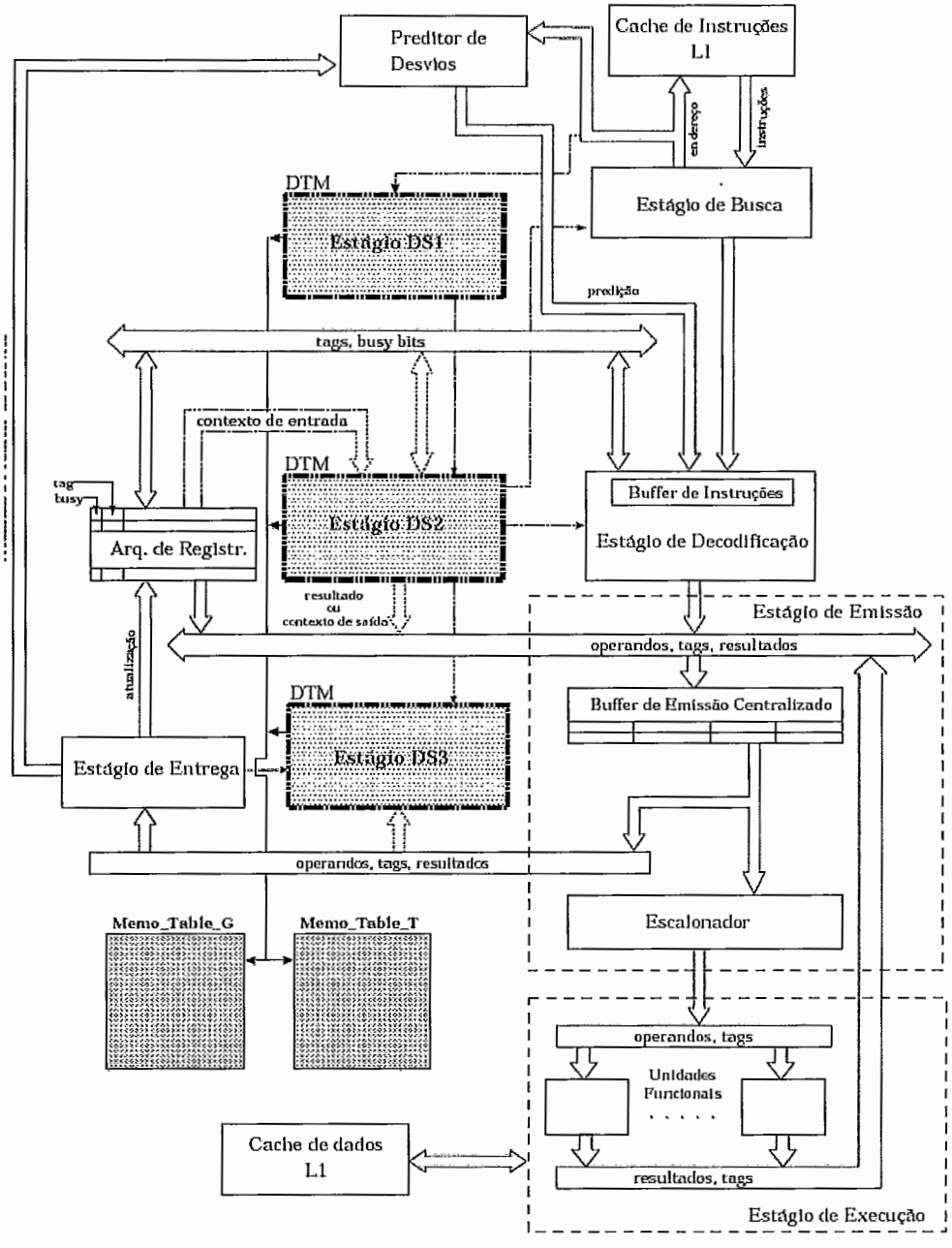


Figura 2.14 – Implementação do Pipeline DTM

Algumas das operações realizadas pelos estágios específicos do *DTM* têm sua latência sobreposta pela latência dos estágios originais da arquitetura, não acarretando em um atraso extra no fluxo convencional devido a sua implementação. Porém, a operação de comparação dos valores armazenados no arquivo de registradores, que compõem o mapa contextual de entrada, com os valores armazenados na *Memo*

Table T tem a sua latência visível e, dependendo da freqüência de clock do processador, um estágio extra no pipeline pode se fazer necessário para os acessos àquela tabela.

As primeiras experiências com uma implementação *DTM* apontaram um ganho de desempenho em relação a um esquema S_{n+d} , quando ambos possuíam a mesma capacidade de armazenamento de informações, enquanto que o *DTM* superou em muito um esquema de Reuso de Blocos, mesmo quando este tinha o dobro da capacidade de armazenamento de informações (COSTA; FRANÇA, *et al*, 2000).

2.7 Reuso através da Especulação de *Traces* (RST)

Apesar do ganho significativo gerado pela implementação do mecanismo *DTM*, percebeu-se que muito do seu potencial ainda não tinha sido totalmente aproveitado, visto que muitos *traces* deixavam de ser reusados pelo fato dos valores do seus operandos de entrada não estarem prontos no momento do teste de reuso (*early validation*). Em média, constatou-se que 68% dos *traces* reusáveis não eram reusados por este motivo (PILLA; 2001, 2002, 2003a, 2003b). Esta média tende a crescer a medida que o pipeline da arquitetura torna-se mais profundo, já que os *traces* precisarão de mais ciclos para serem criados e aumenta a possibilidade de que mais entradas estejam indisponíveis no momento do teste de reuso. Para reverter este quadro, foi introduzida uma nova técnica – *RST* (*Reuse Through Speculation on Traces*) - que integra a predição de valores ao reuso de *traces* (PILLA; 2001, 2002, 2003a, 2003b).

A principal desvantagem do emprego isolado da técnica de reuso de valor está na espera obrigatória por todos os valores dos operandos de entrada para a efetuação do teste de reuso, gastando assim muitos dos ciclos que poderiam ser poupados pelo reuso de instruções.

A técnica de previsão de valores pode tornar transparentes os limites impostos pelas dependências de dados verdadeiras, pois instruções que apresentam esta relação podem ser executadas paralelamente. Sua maior desvantagem está penalidade da recuperação do contexto arquitetural quando uma previsão é falha. Quanto mais profundo o pipeline, mais impactante torna-se esta penalidade. Outro ponto negativo desta técnica é que, com a implicação no aumento da demanda pelos recursos da arquitetura, instruções em execução após uma previsão equivocada podem evitar que instruções válidas entrem em execução.

A técnica *RST* combina as principais vantagens das técnicas de reuso e de previsão de valores, visando impulsionar ainda mais o reuso de *traces* e esconder relações de dependências de dados verdadeiras entre instruções. Nesta abordagem, *traces* podem ser reusados regularmente, estando os valores dos seus operandos de entrada disponíveis no momento do teste de reuso, ou especulativamente, do caso contrário. Assim, é possível explorar não somente a redundância dos *traces*, mas como também sua previsibilidade (PILLA; 2001, 2002, 2003a, 2003b).

2.7.1 A Especulação no *RST*

No *RST*, temos portanto tanto o teste de reuso (*early validation*) quanto o teste de previsão (*late validation*). Na identificação de uma oportunidade de reuso, as entradas do trace são comparadas com as armazenadas na tabela de memorização e no caso de haverem entradas ainda indisponíveis no momento do teste de reuso estas terão seus valores especulados pelo *RST*. Os valores indisponíveis, no momento do teste de reuso, dos *traces* reusados especulativamente serão verificados posteriormente no teste de previsão.

Outro ponto relevante da implementação é que não requere uma quantidade significativa extra de hardware em relação a técnicas que não empregam a especulação, não adicionando qualquer tabela. Como os valores já estão armazenados na *Memo Table T* para o teste de reuso, os mesmos podem ser usados também para a previsão. O número de acessos à tabela de memorização global também não é acrescido em comparação ao *DTM*, uma vez que as consultas sempre são feitas para o teste de reuso, porém muitas vezes inutilmente devido a indisponibilidade dos valores de alguns operandos de entrada. O *RST* busca explorar também o reuso nestes casos, usando a previsão de valores (PILLA; 2001, 2002, 2003a, 2003b).

Caso um trace seja especulativamente reusado no *RST*, os valores de saída, armazenados no mapa contextual de saída do trace na *Memo Table T*, são encaminhados diretamente ao estágio de *Writeback*, alimentando instruções que estavam esperando por tais valores. O trace reusado faz um *bypass* pelos estágios de Despacho, Emissão e Execução, em um único ciclo, logo após ter sido validado como reusável. Portanto, em alguns casos, o reuso especulativo ajuda na diminuição da requisição pelas unidades funcionais, mesmo no caso de uma previsão incorreta, quando há o descarte dos valores de saída do trace e a busca pela próxima instrução é redirecionada.

Para o reuso especulativo, há a necessidade da adoção de um mecanismo de confiança e uma técnica para a previsão de valores eficientes. Quanto à previsão de valores, o *RST* faz uso da técnica *Last n-Value Prediction*, uma variação de *Last Value Prediction*, a qual usa uma tabela de predição de valores indexada pelo *PC* da instrução, tendo cada entrada associada ao valor a ser usado especulativamente. No caso do *RST*, utiliza-se a própria *Memo Table T* , que é indexada também pelo *PC* da instrução inicial do trace, e utiliza-se os valores armazenados na tabela para o reuso especulativo dos valores dos operandos de entrada que estejam indisponíveis no momento do teste de

reuso. Dependendo da associatividade *n* da *Memo Table T*, podemos ter várias entradas de trace reusáveis para um mesmo *PC*, portanto o *RST* opta pelos valores da entrada com base na informação de qual foi menos recentemente usada, a qual é retida na tabela junto ao trace.

Outra técnica testada com o *RST* foi a *Stride Prediction* (Previsão por Strides), que torna possível a previsão de um valor, mesmo que este não tenha sido visto anteriormente, baseando-se em valores anteriores que seguem um determinado padrão que o preditor tenha sido capaz de identificar. Além do valor, é também armazenado o *stride* (diferença entre dois valores consecutivos), que será usado para a eventual especulação do próximo valor. Para tal, o *RST* precisa comparar os valores de dois contextos de entrada consecutivos de um trace para a busca por um *stride*. Esta técnica é bastante útil para a previsão de valores em laços. Confirmando-se a existência de um *stride*, o mesmo é guardado e comparado com a próxima instância daquele trace para verificar se permanece constante na próxima iteração.

Além da informação de *stride* ser importante para um bom desempenho no reuso especulativo de *traces*, a mesma pode ser obtida também a partir do emprego da técnica *Last n-Value Prediction* quando os *traces* são re-acessados. Apesar da técnica de Previsão por Strides ter oferecido um pequeno ganho de desempenho sobre a *Last n-Value Prediction*, o custo para a sua implementação é alto devido ao hardware extra necessário para implementá-la (PILLA; NAVAUX, *et al*, 2004).

Foi realizada uma série de testes para a comprovação da importância de um mecanismo de confiança bastante eficiente e preciso para o proveito do ganho de desempenho oferecido pela especulação no reuso de *traces*. Um mecanismo com baixo nível de confiança, mas com bom desempenho, indicaria a necessidade de uma implementação simples para evitar previsões incorretas das entradas de *traces*

especulados. Para determinar qual nível de confiança seria necessário, para atingir uma bom desempenho no *RST*, variou-se o percentual do nível de precisão do mecanismo comparando-o com o desempenho da arquitetura. Concluiu-se que o mecanismo de confiança deva ter uma precisão no *range* de 90% a 99% para que se obtenha bons resultados de desempenho provenientes da especulação (PILLA; NAVAUX, *et al*, 2003).

No *RST*, além do reuso de *traces*, há também a possibilidade do reuso de uma instrução isoladamente, porém inicialmente apenas para *traces* foi adotada a estratégia especulativa. Isto pelos simples fato dos *traces* encapsularem várias instruções e possivelmente caminhos críticos, viabilizando uma melhora maior no desempenho quando reusados, comparando-se ao reuso de uma única instrução.

2.7.2 Comparativo entre *RST* e *DTM*

A implementação *RST* adota as mesmas tabelas – *Memo Table G* e *Memo Table T* - introduzidas pelo mecanismo *DTM*. Porém, além da diferença entre as duas abordagens quanto a identificação de *traces* reusáveis, o *RST* também oferece um segundo modo para a construção de *traces* além do já implementado pelo *DTM*. No *DTM*, todas as instruções válidas, pertencentes ao domínio, tinham suas informações armazenadas primeiramente em *Memo Table G* para que depois pudessem constituir um trace. O trace, no *DTM*, era constituído apenas por instruções redundantes, que já possuíam uma entrada na tabela de memorização global. O *RST* permite a inclusão de instruções não redundantes no trace, durante seu processo de construção, desde que pertençam ao domínio de instruções válidas. O resultado obtido foi um maior número de *traces* identificados e reusáveis (PILLA; 2001, 2002, 2003a, 2003b).

O *RST* então oferece duas políticas para a formação de *traces* e o mecanismo é flexível quanto a escolha e alternância entre estas. A política conhecida como *reused-only*, já implementada no *DTM*, permite que apenas instruções com entrada em *Memo Table G* possam fazer parte do trace. A política *reusable mode* permite que instruções ainda não marcadas como redundantes possam integrar o trace. Esta segunda tem a desvantagem de permitir que alguns *traces* sejam vitimados (retirados da tabela) da *Memo Table T* antes de serem reusados.

Em comparação ao *DTM*, os *traces* são formados mais rapidamente na implementação *RST* com a política *reusable mode*, pois as instruções não precisam ter suas informações primeiramente armazenadas na *Memo Table G* para que somente depois possam integrar um trace. Esta característica também aumenta o número de instruções candidatas potenciais à formação de *traces*. Em contrapartida, alguns *traces* formados podem não apresentar tanta redundância e a *Memo Table T* pode ficar bastante povoada por estes *traces* (PILLA; 2002, 2003a, 2003b).

2.7.3 A Implementação do *RST*

A arquitetura *RST* é composta por um pipeline de quatro estágios, que opera em paralelo ao pipeline da arquitetura substrato, como exibido pela Figura 2.4 (PILLA; 2001, 2002, 2003a, 2003b), evitando impactos na complexidade do caminho crítico das instruções.

O primeiro estágio, *RS1*, é responsável pela busca por instâncias candidatas ao reuso nas tabelas de memorização, com base no valor de *PC* recebido, e as encaminha para o estágio *RS2*. O número máximo de candidatas enviadas para *RS2* depende da associatividade das tabelas. O estágio *RS1* também recebe de *RS4* informações de instruções e *traces* candidatos ao reusáveis, a serem armazenados nas tabelas de memorização caso nenhum entrada coincidente seja encontrada.

O estágio *RS2* compara as entradas da instrução ou trace candidato a reuso com os valores retidos pelo arquivo de registradores. Caso haja valores de entrada ainda indisponíveis no momento do teste de reuso, um mecanismo de confiança é acessado para checar se o valor deverá ser previsto ou não, de modo a evitar ao máximo as previsões incorretas. Quanto maior a profundidade do pipeline, mais necessária se faz a presença de um mecanismo de confiança eficiente na arquitetura. As possíveis saídas deste estágio são: os valores de saída de um trace reusado e um novo valor de *PC* para redirecionamento do fluxo de execução; ou um trace reusado especulativamente e seus valores de entrada especulados; ou o valor de saída de uma instrução reusada e um novo valor de *PC* para redirecionamento do fluxo de execução.

O estágio *RS3* é onde ocorre o teste da previsão (*late validation, misprediction test)*, comparando os valores reais com os especulados. O mecanismo implementado neste estágio é o mesmo já presente nas arquiteturas para a tarefa de previsão de desvios. Este estágio também armazena as novas previsões e as respectivas informações para recuperação em caso de falha. Havendo sucesso nas previsões *RS3* efetiva o *Commit* do trace, caso contrário um novo valor de *PC* é encaminhado ao estágio de *Busca*. De qualquer forma, os resultados das previsões são enviados para o estágio *RS2* para a atualização da tabela do mecanismo de confiança.

O estágio *RS4* é responsável pela identificação de instruções reusáveis e a formação de *traces*, conforme a política escolhida (*reused-only* ou *reusable*). Após a construção de um trace, este é encaminhado ao estágio *RS1*. As instruções seguintes a uma previsão mal sucedida são descartadas nesta fase, e a busca é redirecionada para o endereço no qual o trace iniciou-se.

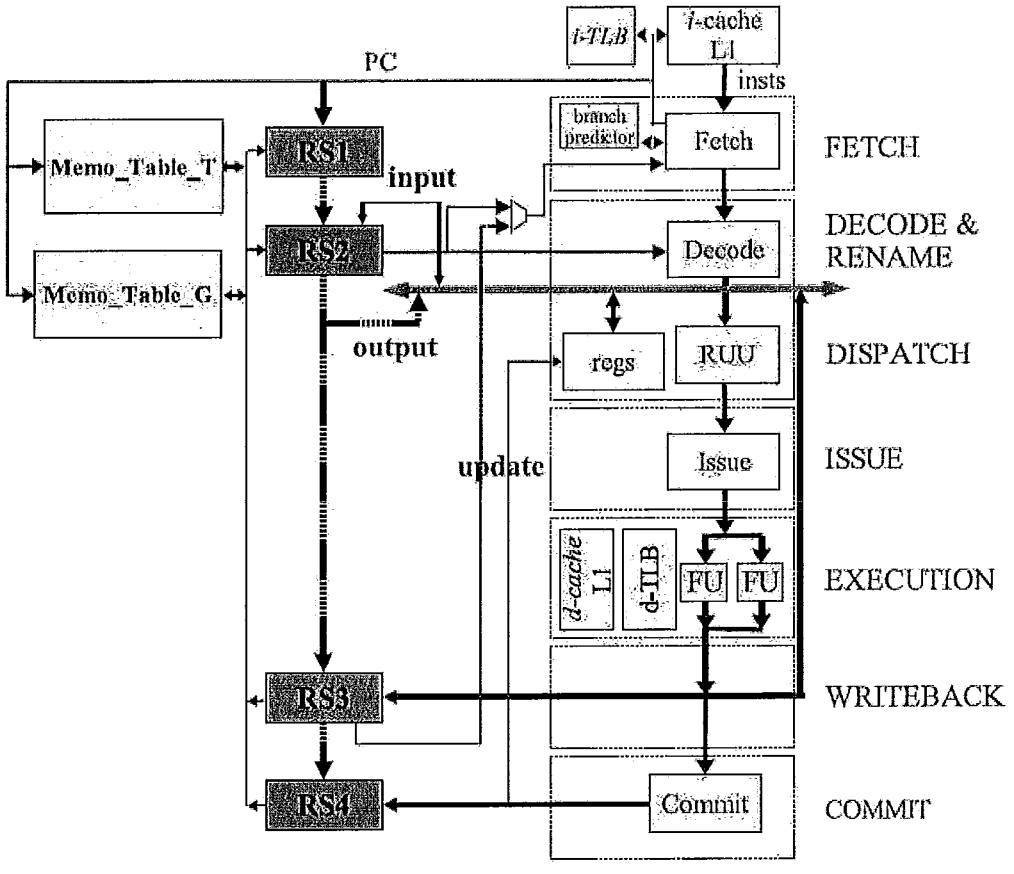


Figura 2.15 – Implementação do Pipeline RST

2.7.4 Optimizações da Arquitetura *RST*

Para a redução da complexidade da implementação *RST*, mas mantendo seu potencial, foram realizados experimentos para obter os números e tamanhos ideais a serem considerados para os contextos de entrada e saída dos *traces* e das tabelas de memorização. O número de registradores de entrada e saída de um trace influencia o tamanho da entrada na tabela de memorização necessário para armazená-lo, portanto também o tamanho da própria tabela.

Constatou-se que, para a maioria dos *traces*, os contextos de entrada e saída estão restritos a 4 valores de registradores de entrada, 4 valores de registradores de saída e 3 referências à memória. Como o tamanho da tabela de memorização acaba por

determinar quais *traces* serão capturados e reusados, a melhor combinação encontrada, para a redução do custo do hardware e manutenabilidade do desempenho, consistiu em uma *Memo Table T* relativamente pequena, com 512 entradas, e uma *Memo Table G* com 4K entradas. Esta configuração apresentou bom desempenho quando comparada com configurações que utilizavam tabelas significativamente maiores (PILLA; NAVAUX, *et al*, 2003).

Outra questão considerada foi a largura de banda requisitada entre os estágios *RS1* e *RS2* do pipeline da arquitetura *RST*. Este tráfego depende diretamente do número de *traces*, instruções, tamanho dos mapas contextuais de entrada e saída, e da associatividade das tabelas de memorização. Por exemplo, usando tabelas associativas de 4 vias, uma consulta de *RS1* pode retornar até 4 *traces* e 4 instruções candidatas ao reuso para o teste de reuso feito em *RS2*. Para reduzir o tráfego de informação entre estes estágios, buscou-se tanto diminuir o número de instruções e *traces* candidatos ao reuso quanto o tamanho dos contextos de entrada e saída dos *traces*. O primeiro passo nesta direção foi a redução do número de tabelas de memorização de 2 para 1, ou seja, a unificação de *Memo Table G* e *Memo Table T* em uma única tabela, com tamanho equivalente à soma dos tamanhos das duas tabelas originais, que seria utilizada para o reuso de instruções e *traces*. Esta mudança diminuiu o número de candidatos, gerou uma sutil queda de desempenho, mas ainda não foi o suficiente para atenuar significativamente o tráfego entre *RS1* e *RS2*. O segundo passo foi adotar a associatividade com mapeamento direto nesta tabela unificada, limitando cada *PC* a uma única entrada de candidato ao reuso, que gerou uma redução sensível de 75% na largura de banda requisitada e um decréscimo de desempenho, a qual em média ainda manteve-se no patamar da configuração anterior com associatividade de 4 vias. A terceira medida adotada foi a redução do tamanho dos contextos de entrada e saída do

trace, tornando-os equivalentes ao de uma instrução, ou seja, 2 valores de entrada e 1 valor de saída. Esta configuração duplicou o número de entradas disponíveis na tabela unificada, simplificou os requisitos para o reuso de instruções e *traces*, e viabilizou a especulação de instruções isoladas além dos *traces*, já que uma instrução ocupa o mesmo espaço na tabela que um trace. Estas alterações reduziram em mais de 8 vezes o tráfego entre os estágios *RS1* e *RS2*, em comparação a configuração original, ainda mantendo uma média de 21% de ganho de desempenho do *RST* sobre uma arquitetura sem reuso de valor (PILLA; FRANÇA, *et al*, 2006).

Embora a arquitetura com 2 tabelas de memorização associativas de 4 vias apresente um *speedup* (ganho de desempenho) maior em relação à arquitetura com tabela unificada, entradas reduzidas e mapeamento direto, mais complexa é a implementação de suas tabelas e maior é a largura de banda necessária para a transferência dos dados das tabelas para o estágio do teste de reuso. Com uma perda de apenas 4% no *speedup*, a arquitetura simplificada aparece como a melhor solução para implementação do *RST*, pois apesar de capturar *traces* com menor comprimento em média, detecta um número maior de *traces* reusáveis e mantém a resolução das cadeias de dependências de dados (PILLA; FRANÇA, *et al*, 2007).

2.8 Processadores com Memorização Automática

A Memorização Dinâmica tem se apresentado como uma técnica eficaz para superar muitas das limitações encontradas pelas técnicas de paralelismo a nível de instruções, tais como a questão do *memory wall* ("Barreira de Memória") , decorrente da diferença crescente de desempenho entre o sistema de processamento e o de memória, e que restringe a largura de banda para a emissão de instruções.

Baseando-se nos mesmos conceitos introduzidos pelo mecanismo *DTM* , e empregando uma técnica para a especulação de valores, tal como o *RST*, foi apresentada a proposta de um processador com memorização automática e dinâmica, capaz de realizar o reuso de conjuntos de instruções sem a necessidade de recompilação do código original (TSUMARA; SUZUKI, *et al*, 2007).

Este processador detecta funções e laços em múltiplos níveis sem o auxílio de instruções adicionais inseridas pelo compilador. As regiões reusáveis tem suas fronteiras identificadas dinâmicamente pelas instruções de chamada e retorno de funções, ou pelas instruções de desvio e do endereço-alvo deste no caso dos laços. A memorização desta regiões é feita automaticamente em uma tabela, implementada em memória *on-chip* ternária do tipo CAM (*Contents Addressable Memory*), denominada *MemoTbl* e que armazena as entradas e saídas das regiões armazenadas para posterior reuso.

A identificação das variáveis de entrada e saída destas regiões reusáveis, no caso das funções, depende diretamente da segmentação feita pelo sistema operacional, na memória principal (*área de dados* e *área de pilha*), quanto ao armazenamento de variáveis globais e locais. Tomando como base o valor do *stack pointer*, é feita a distinção entre as variáveis locais instanciadas nos vários níveis. No caso dos laços, não há como determinar quais variáveis são locais, portanto todas as variáveis referenciadas no laço são consideradas como variáveis de entrada daquela região reusável.

O sistema de memorização é composto de um mecanismo de memorização, um pequeno *buffer* (*MemoBuf*) para a escrita dos valores de entrada e saída da região reusável ao longo da sua execução, e da tabela *MemoTbl*. Após a execução de uma região considerada reusável, os contextos de entrada e saída armazenados temporariamente no *MemoBuf* são transferidos para uma entrada na *MemoTbl*.

Geralmente, as regiões reusáveis apresentam-se aninhadas em um programa, tornando-se necessário que este processador seja capaz de armazenar as entradas e saídas de regiões aninhadas, simultâneamente, usando a estrutura *MemoBuf* como uma espécie de pilha, na qual cada entrada contém informações de entrada/saída de uma região. Ao término da execução da região, a sua respectiva entrada em *MemoBuf* é transferida para a *MemoTbl*.

Quando inicia-se a execução de uma região considerada reusável, o processador consulta *MemoTbl* para o teste de reuso, comparando os valores de entrada do contexto arquitetural com os anteriormente armazenados na tabela. Havendo equivalência, o mecanismo de memorização encaminha os valores de saída, retidos na tabela, para a *cache* e os registradores, omitindo a execução daquela região. No caso de retorno negativo do teste de reuso, a execução da região procede normalmente e o processador faz a memorização do seu contexto de entrada e saída em paralelo.

Além do reuso de valor, a técnica de previsão de valores também é utilizada através da execução em vários *cores* especulativos, paralelamente ao *core* principal no qual a região reusável estará em execução, buscando explorar o as vantagens oferecidas pelo *design multi-core* dos microprocessadores atuais. Um exemplo típico de onde é preciso usar este artifício são as iterações de um laço, cuja variável de controle faz parte do contexto de entrada da região, que acabaria por nunca se beneficiar da memorização caso contrário.

Este modelo é conhecido como *Speculative Multi-Threading* (*SpMT*) e, diferentemente da implementação convencional do mecanismo de especulação, o *core* principal não sofre a penalidade originada por uma previsão mal sucedida. Enquanto o *core* principal executa o código da região reusável, os *cores* especulativos executam esta mesma região concomitantemente mas com valores de entrada especulados, com

base no último valor apresentado ou em um padrão de *strides* identificado. Cada *core* especulativo executa a região reusável com um valor de entrada previsto, diferente dos usados pelos demais *cores*, e armazena o resultado obtido na *MemoTbl*, aumentando assim a probabilidade de sucesso da previsão. Caso um valor previsto tenha sido comprovado como correto, o *core* principal já poderá encontrar o próximo resultado na tabela de memorização.

2.9 Processadores *Contrail*

A arquitetura do tipo *Contrail* também baseia-se no paralelismo no nível de *threads*, tendo como foco a redução no consumo de energia e manutenção do desempenho, ao invés do aumento da última como nas demais implementações citadas neste trabalho.

Nos processadores do tipo *Contrail*, a execução da aplicação é dividida em duas *threads*. A *thread* de especulação é responsável pelo fluxo de execução principal da aplicação, sendo despachada em unidades funcionais de baixa latência. Nesta *thread*, há o *bypass* de algumas regiões de código, que não são executadas, conforme indicação de um preditor de valor a nível de *traces*. A segunda *thread*, chamada de *thread* de verificação, faz a validação de cada previsão de valor e tem o seu despacho realizado em unidades funcionais mais lentas. O intuito desta divisão é a transformação dos caminhos críticos em não-críticos, através da implementação do preditor de valores a nível de *traces*, fazendo com que instruções consideradas como não-críticas, capturadas por este preditor, sejam executadas em uma *thread* à parte, em unidades funcionais com maior latência e com freqüência de clock reduzida. Estas características possibilitam uma redução no nível de energia necessária para a operação. Desta forma, a *thread* de especulação está com seu fluxo de execução sempre à frente da *thread* de verificação, o que originou o nome adotado para esta arquitetura (KOUSHIRO; SATO, *et al*, 2003).

Para compensar o consumo de energia adicional que o preditor de valores a nível de *traces* imporia, foram realizadas adaptações ao projeto original deste preditor. A estratégia é usar uma tabela para o armazenamento de *traces* mais enxuta, retendo menos informações, e usar um preditor de valores a nível de instruções, compondo assim o denominado Preditor de Valores a Nível de *Traces* Desacoplado. Ao invés de uma única consulta, em um único ciclo de clock, para a previsão de todos os valores do trace, este mecanismo realiza a previsão de cada valor do trace seqüencialmente, requerendo vários ciclos de clock para a previsão do contexto de entrada completo daquele trace. A consulta é feita à tabela de *traces*, a qual armazena os valores de *PC* das instruções que referenciam registradores que integram o contexto de entrada do trace. O preditor de valores a nível de instrução é então invocado várias vezes, uma para cada entrada a ser especulada, simplificando e reduzindo o custo de hardware deste mecanismo.

Esta proposta apresentou uma economia de energia na média de 40%, mantendo o mesmo nível de desempenho, comparada a uma arquitetura convencional. Constatou-se também que a inovação do preditor desacoplado oferece a mesma precisão de um preditor convencional, mas com uma redução aproximada de 70% no custo de hardware (KOUSHIRO; SATO, *et al*, 2003).

CAPÍTULO 3 – O Impacto do Reuso de Traços internos a Laços

Neste capítulo será apresentado o estudo feito sobre as características e o perfil dos *traces* reusáveis, com base nos *logs* de execução gerados pelo simulador *sim-rst*, adaptado do conjunto *Simple Scalar Suite 3.0* para a emulação de uma arquitetura *MIPS* com a implementação do mecanismo *RST*.

3.1 A Estratégia da Análise

O objetivo desta análise teve como foco a identificação de um padrão na seqüência de instruções que precediam os *traces* capturados pelo mecanismo *RST*. A versão *sim-rst* do simulador é capaz de gerar um log da seqüência de instruções executadas, em modo estatístico (não-funcional), para o programa passado como parâmetro, evidenciando quais instruções foram eventualmente reusadas de forma isolada (*flag* de reuso com valor *1*), ou em *traces* (*flag* de reuso com valor *2,* marcando início de trace, ou *3*, identificando meio ou fim de trace), e quais tiveram de ser executadas (*flag* de reuso igual a *0*), ou seja, passaram pelo caminho de dados completo para obtenção do seu resultado.

Como foco inicial deste estudo, foram gerados logs para um subconjunto dos *benchmarks* do *SPECint2000* referentes à execução de 100.000 instruções pelo simulador *sim-rst*, após a execução das 150 milhões de instruções iniciais do programa, sendo 100 milhões destas em modo funcional e as demais 50 milhões em modo estatístico. A captura de dados após a execução de um número inicial de 50 milhões de instruções buscou garantir o povoamento das tabelas de reuso até o momento em que se iniciaria a coleta dos dados em *log*.

No *log* de execução (Figura 3.1), além da *flag* de reuso, há um contador que acompanha seqüência de instruções executadas, o *PC* (*Program Counter*), o

mnemônico e a representação em assembly de cada instrução. Estas informações foram usadas para o rastreamento do padrão dos *paths* que antecediam aos *traces* capturados e identificados no log.

```
TRACE_START
mm0 0      0 :: 0x00405ec8     addu      r9,r7,r2
mm0 0      1 :: 0x00405ed0     bne       r9,r0,0x405f10
mm0 0      2 :: 0x00405f10     bltz      r9,0x405f28
mm0 0      3 :: 0x00405f28     slt       r1,r16,r17
mm0 0      4 :: 0x00405f30     bne       r1,r0,0x405f90
mm0 0      5 :: 0x00405f38     sll       r2,r17,2
mm0 0      6 :: 0x00405f40     addu      r12,r3,r2
mm0 0      7 :: 0x00405f48     lw        r13,0(r12)
mm0 0      8 :: 0x00405f50     sll       r2,r16,2
mm0 0      9 :: 0x00405f58     addu      r10,r3,r2
mm0 0     10 :: 0x00405f60     lw        r11,0(r10)
mm0 0     11 :: 0x00405f68     sw        r11,0(r12)
mm0 0     12 :: 0x00405f70     sw        r13,0(r10)
mm0 0     13 :: 0x00405f78     addiu     r17,r17,1
mm0 0     14 :: 0x00405f80     addiu     r16,r16,-1
mm0 1     15 :: 0x00405f88     j         0x405de8
mm0 0     16 :: 0x00405de8     sll       r2,r18,2
mm0 0     17 :: 0x00405df0     addu      r10,r3,r2
.
.
mm0 0    909 :: 0x00405f78     addiu     r17,r17,1
mm0 0    910 :: 0x00405f80     addiu     r16,r16,-1
mm0 1    911 :: 0x00405f88     j         0x405de8
mm0 0    912 :: 0x00405de8     sll       r2,r18,2
mm0 0    913 :: 0x00405df0     addu      r10,r3,r2
mm0 0    914 :: 0x00405df8     slt       r1,r16,r17
mm0 0    915 :: 0x00405e00     bne       r1,r0,0x405e88
mm0 0    916 :: 0x00405e08     sll       r2,r17,2
mm0 0    917 :: 0x00405e10     addu      r12,r3,r2
mm0 0    918 :: 0x00405e18     lw        r13,0(r12)
mm0 0    919 :: 0x00405e20     lbu       r2,(r8+r13)
mm0 0    920 :: 0x00405e28     addu      r9,r7,r2
mm0 2    921 :: 0x00405e30     bne       r9,r0,0x405e70
mm0 3    922 :: 0x00405e70     bgtz      r9,0x405e88
mm0 0    923 :: 0x00405e78     addiu     r17,r17,1
mm0 1    924 :: 0x00405e80     j         0x405df8
mm0 0    925 :: 0x00405df8     slt       r1,r16,r17
mm0 0    926 :: 0x00405e00     bne       r1,r0,0x405e88
```

Figura 3.1 – Parte do log de execução gerado pelo *sim-rst* para o programa *BZIP2*

3.2 Os Laços como um Padrão

Para a interpretação dos *logs* gerados, foi desenvolvido um código tipo *script* na linguagem *Perl*, que evoluiu em 3 versões para o acompanhamento da análise. A linguagem *Perl* foi escolhida pela facilidade encontrada na sua sintaxe para a manipulação de padrões de *strings* e expressões regulares, surgindo como a opção mais apropriada para o processamento dos arquivos de *log* gerados pelo simulador.

O primeiro passo consistiu em contabilizar quantos *traces* foram capturados na execução de cada benchmark, agrupando-os pelo *PC* da instrução de início do trace. De forma a focalizar nos *traces* mais reincidentes, e portanto mais relevantes, foi feita uma listagem destes PC's por odem decrescente do número de ocorrências.

Para cada *PC* desta listagem, percorreu-se o log na busca por ocorrências de início de *traces* com aquele *PC* (*flag* de reuso igual a *2*), e em seguida, comparando-se em pares as seqüências de instruções antecedentes a estes *traces*. Para tal, cada seqüência de instrução precedente a uma trace foi armazenada em vetor, identificado como um bloco, e cada *PC* inicial de trace estava associado a uma relação destes vetores, através de uma tabela *hash,* cuja chave era o *PC*. A comparação de um par, ou seja, entre dois vetores de seqüências de instruções que antecediam *traces* iniciados por uma mesmo *PC*, terminava no momento em que se encontrava uma instrução diferente, percorrendo-se no log o caminho anterior a instrução do início do trace.

```
REVERSE 0x004145a8
REVERSE 0x00414650
REVERSE 0x00417e30
REVERSE 0x00413968
REVERSE 0x00413a58
REVERSE 0x00417d98
REVERSE 0x004142c0
REVERSE 0x00415418
REVERSE 0x004057c0
REVERSE 0x00405880
REVERSE 0x0040c8d8
14 match(es) for PC 0x004145a8 (4277672) among blocks 0
(line 14) and 1 (line 42)
14 match(es) for PC 0x004145a8 (4277672) among blocks 0
(line 14) and 2 (line 398)
14 match(es) for PC 0x004145a8 (4277672) among blocks 0
(line 14) and 3 (line 426)
...
27 match(es) for PC 0x004145a8 (4277672) among blocks 22
(line 3960) and 459 (line 82923)
27 match(es) for PC 0x004145a8 (4277672) among blocks 22
(line 3960) and 460 (line 82951)
27 match(es) for PC 0x004145a8 (4277672) among blocks 22
(line 3960) and 461 (line 82979)
...
128 match(es) for PC 0x0040c8d8 (4245720) among blocks 43
(line 77175) and 53 (line 94885)
128 match(es) for PC 0x0040c8d8 (4245720) among blocks 43
(line 77175) and 54 (line 96666)
128 match(es) for PC 0x0040c8d8 (4245720) among blocks 43
(line 77175) and 55 (line 98447)
374 match(es) for PC 0x0040c8d8 (4245720) among blocks 44
(line 78911) and 45 (line 80692)
374 match(es) for PC 0x0040c8d8 (4245720) among blocks 44
(line 78911) and 46 (line 82473)
374 match(es) for PC 0x0040c8d8 (4245720) among blocks 44
(line 78911) and 47 (line 84254)
```

Figura 3.2 – Resultado gerado pelo Script para a interpretação do log de execução do benchmark *MCF*. Identificação dos PC's iniciais de *traces* e comparação entre as seqüências de instruções precedentes a cada ocorrência de trace iniciado por um determinado PC.

A repetição de um número fixo de combinações de instruções (*matches*), na comparação entre os blocos que antecediam *traces* iniciados por um mesmo *PC* (Figura 3.2), indicou a existência de certo padrão, característico a estruturas amplamente utilizadas nos códigos de programas, os laços.

Esta constatação redirecionou o estudo de forma a focar no reconhecimento da existência de laços que estivessem englobando os *traces* capturados, assim

caracterizando um padrão na incidência destas seqüências de instruções redundantes. Para este fim, foram realizadas novas alterações no Script de interpretação, de forma que este identificasse instruções de início e fim de iterações de laços e apontasse a ocorrência de *traces* capturados nestas regiões.

```
Analyzing PC 0x004efe90

PC 0x004efe90 reused  loop de inicio  0x004efd30 e fim 0x004efe90, size 55
Iteração que se inicia na linha 467
2 instruções do trace dentro do loop
53 instruções do loop precedentes ao trace

PC 0x004efe90 reused  loop de inicio  0x004efd30 e fim 0x004efe90, size 55
Iteração que se inicia na linha 4206
2 instruções do trace dentro do loop
53 instruções do loop precedentes ao trace

Atingiu  PC 0x004efe90, mas não se trata do mesmo trace

PC 0x004efe90 reused  loop de inicio  0x004efd30 e fim 0x004efe90, size 55
Iteração que se inicia na linha 8782
2 instruções do trace dentro do loop
53 instruções do loop precedentes ao trace

PC 0x004efe90 reused  loop de inicio  0x004efd30 e fim 0x004efe90, size 22
Iteração que se inicia na linha 14784
2 instruções do trace dentro do loop
20 instruções do loop precedentes ao trace

PC 0x004efe90 reused  loop de inicio  0x004efd30 e fim 0x004efe90, size 22
Iteração que se inicia na linha 15455
2 instruções do trace dentro do loop
20 instruções do loop precedentes ao trace
```

Figura 3.3 – Identificação de laços que continham *traces* capturados no log de execução do benchmark GCC.

A nova versão do Script gerou resultados (Figura 3.3) que comprovaram a alta incidência de *traces* nas iterações de laços. Para cada ocorrência de trace iniciado por um determinado *PC*, percorreu-se pelas instruções adiante no *log* até encontrar-se um *PC* cujo valor era menor que o do *PC* da instrução anterior, indicando um salto para uma região de código já anteriormente executada, e que poderia tratar-se potencialmente

de um laço. Desta forma, aqueles *PC's* eram caracterizados como as instruções de final e início de laço. Em seguida, percorreu-se o log de execução adiante até que fosse encontrada uma instrução de *PC* igual a do início daquele trace, certificando-se de que o mesmo encontrava-se dentro da iteração de um laço. Uma vez identificada a iteração de laço, o contador do log de execução foi usado para contabilizar o número de instruções pertencentes ao *trace* que estavam dentro do laço e quantas instruções do próprio laço precediam a instrução inicial do trace.

3.3 As Versões para um Reuso Localizado

O indício dos laços apresentarem-se como uma das principais regiões de código reusável remeteu esta análise a identificar-se com a proposta de *design* de um processador com memorização automática de *loops* (laços) e funções, apresentada por TSUMARA; SUZUKI, *et al*, 2007 e referenciada no capítulo 2 deste trabalho. Este projeto consistia de um processador com memorização automática e dinâmica, capaz de detectar trechos de códigos de laços e funções e armazená-los como regiões de código passíveis de reuso, além de empregar a execução em paralelo em múltiplos *cores* para uma abordagem especulativa.

Vislumbrando-se a estreita relação entre os *traces* capturados e os laços existentes no código dos programas, adotando uma estratégia similar a do processador com memorização automática, foram feitas adaptações no código do simulador *sim-rst* (PILLA; 2001, 2002, 2003a, 2003b). de forma a limitar o emprego das técnicas de reuso apenas às regiões dos laços. Isto tornou-se possível devido a parametrização já existente no simulador *sim-rst*, que permite a realização de simulações mediante a ativação ou desativação dos mecanismos de reuso e especulação.

Nesta versão modificada, denominada *sim-rst-loop*, a habilitação o mecanismo de reuso é feita dinamicamente, quando da identificação de um laço no estágio de *commit*, verificando se o *PC* da instrução posterior é inferior ao da instrução corrente e se esta é uma operação de salto condicional (*BEQ* ou *BNE*) ou incondicional (*J*). Ao detectar o término do laço, a *flag* do mecanismo de reuso volta a ser desativada. A identificação de um laço só é possível ao término da sua primeira iteração, quando então é ativado o mecanismo de reuso.

Em contrapartida, também se adaptou a versão *RST* para que realizasse o reuso de *traces* capturados apenas fora das estruturas de laços. Esta versão foi denominada *sim-rst-outofloop*.

A implementação destas duas versões adaptadas tem por objetivo validar a tese de que o código residente em laços é, na maioria dos casos, o grande responsável pela geração de seqüências de instruções redundantes e reusáveis. A adoção de uma estratégia para o reuso localizado visa concentrar esforços em uma região de código menos abrangente e mais valiosa.

CAPÍTULO 4 – Base Experimental, Resultados e Avaliações

Para validar a teoria elaborada na fase de análise, quando se identificou uma forte correlação entre os *traces* capturados e os laços internos do código dos programas, adaptou-se o simulador *sim-rst* (PILLA; 2001, 2002, 2003a, 2003b) para duas versões com escopos distintos: uma para o reuso, pelo mecanismo *RST*, apenas de *traces* localizados internamente a laços do código dos programas, chamada *sim-rst-loop*, e outra para o reuso somente de *traces* externos a estruturas de laços, denominada *sim-rst-outofloop*.

4.1 Ambiente de Simulação

O simulador *sim-rst* é a segunda versão modificada do simulador *sim-outorder* do *SimpleScalar3.0 Tool Suite* (BURGER, D., AUSTIM, T.M., June 1997). A primeira versão contemplava a implementação do mecanismo *DTM* (COSTA; 2001), e serviu como fundamento para o desenvolvimento do *RST* (PILLA; 2001, 2002, 2003a, 2003b).

O pacote de ferramentas do *SimpleScalar* permite a modelagem de aplicações que simulam programas reais em execução sob uma variedade de arquiteturas, sendo capaz de emular o conjunto de instruções do *Alpha*, *PISA* e *ARM*. Neste ambiente, é possível segregar a maior parte das questões arquiteturais da própria implementação do simulador, que se baseia em uma arquitetura multifuncional superescalar com *pipeline*. Para esta análise, utilizou-se o conjunto de instruções *PISA*, a qual consiste em um super conjunto das instruções *MIPS* bastante útil para análises de engenharia computacional.

Para os experimentos executados com as versões *sim-rst*, sim-*rst-loop* e *sim-rst-outofloop*, foi utilizado um subconjunto dos programas de teste do pacote SPECInt2000, a seguir: *BZIP2*, *GCC*, *GZIP*, *MCF*, *PARSER* e *VORTEX*.

Para a realização das simulações foram utilizadas estações *INTEL Dual Core 2.0, 3.2 GHz* sob os sistemas operacionais *Linux Debian 4.0* e *Linux Ubuntu 7.04*.

4.2 Configuração do Ambiente

Para realizar os experimentos com as versões *sim-rst*, *sim-rst-loop* e *sim-rst-outofloop*, visando posteriormente confeccionar com um quadro comparativo de desempenho entre estas, utilizou-se a seguinte configuração básica da arquitetura do simulador, conforme atribuição de valores aos seus parâmetros, relacionados nas Tabelas A.1, A.2, A.3, A.4 e A.5, no Anexo A.

Conforme a parametrização do simulador, é possível determinar o nível de reuso que será empregado na execução dos programas, desde a ausência de qualquer técnica de reuso até o emprego de abordagens mais agressivas como o reuso especulativo. A partir das três versões do simulador (*sim-rst, sim-rst-loop* e *sim-rst-outofloop*) disponíveis, foram concebidas nove configurações distintas de arquiteturas para a realização dos experimentos, através de quatro variações da parametrização quanto à estratégia de reuso, da forma descrita abaixo, seguindo as diretrizes dos trabalhos anteriores (COSTA; 2001, PILLA; 2001, 2002, 2003a, 2003b), quanto a melhor configuração em cada caso.

i) *SIMPLE SCALAR* (original)

Nesta configuração, conforme a Tabela A.6, no Anexo A, o mecanismo de reuso não é utilizado.

ii) *DTM / DTM LOOP*

Nesta configuração, o mecanismo de reuso *DTM* é utilizado, porém apenas quando há disponibilidade dos dados de entrada, ou seja, não há suporte à previsão de valores. As instruções e *traces* redundantes são armazenados em separado, em duas tabelas: *Memo_Table_G* para as instruções e *Memo_Table_T* para os *traces*. A configuração está descrita na Tabela A.7, no Anexo A.

iii) *RST / RST LOOP / RST OUT OF LOOP* com Especulação *Non-Oracle*

Nesta configuração, conforme a Tabela A.8, no Anexo A, o mecanismo de reuso *RST* é utilizado, no qual as instruções e *traces* redundates são armazenados em uma tabela unificada. Há o suporte à previsão de valores de entrada, ainda não prontos no momento do teste de reuso, e esta especulação não é perfeita.

Apesar do parâmetro *dtm:moinst* estar configurado com o valor *true*, as instruções permanecerão a ser reusadas nesta arquitetura, porém retidas na mesma tabela na qual os *traces* são armazenados. Neste caso, haverá uma tabela unificada, que será a Memo_Table_T, para o armazenamento de *traces* e instruções reusáveis. Isto se torna possível com a configuração do parâmetro *-dtm:min_trace_size* com o valor *1*, ou seja, um trace de tamanho igual a 1 é equivalente a uma instrução reusável isoladamente.

Diferentemente do *DTM,* onde apenas instruções redundantes compunham os *traces*, nesta configuração *RST* todas as instruções que pertençam ao domínio de instruções reusáveis podem compor um trace, independentemente de serem ou não redundantes. Esta é uma abordagem ainda mais ousada e estabelecida pelo parâmetro *dtm:allow_reusable.*

iv) *RST / RST LOOP / RST OUT OF LOOP* com Especulação *Oracle*

Nesta configuração, conforme a Tabela A.9, no Anexo A, o mecanismo de reuso *RST* é utilizado, no qual as instruções e *traces* redundantes são armazenados em uma tabela unificada. Há o suporte à previsão de valores de entrada, ainda não prontos no momento do teste de reuso, e esta especulação é perfeita (modo "*oráculo*"), de forma que os valores reais sempre coincidem com os valores anteriormente especulados. Como o modo de especulação é perfeito ("*oráculo*"), os parâmetros do mecanismo de confiança são ignorados.

4.3 Comparação Analítica dos Resultados

Para o propósito de uma análise, comparando-se as arquiteturas apresentadas na seção anterior, foram executadas simulações com os benchmarks do conjunto SPECInt2000 para uma amostra de 1,5 bilhões de instruções. Dos resultados provenientes de cada simulação, cinco foram considerados como mais significativos para este estudo: o *IPC* (número de instruções entregues por ciclo de clock), o número de *traces* capturados em *Memo_Table_T*, o número de acessos à cache de dados e de instruções de primeiro nível (*DL1* e *IL1*), e o número de consultas feitas à tabela de previsão de desvios.

O principal fator levado em consideração foi o *speedup* (aceleração de desempenho) apresentado, resultante da divisão do valor de *IPC* de uma nova versão sobre o da versão adotada como base. Tendo este parâmetro como referência, realizou-se uma série de simulações decorrentes da variação de outros elementos da arquitetura que pudessem impactá-lo, como o tamanho e a associatividade da tabela de reuso de *traces* (*Memo-Table_T*) e o tamanho das caches de dados e de instruções de primeiro nível. A seguir, estes experimentos são apresentados com os seus resultados e as devidas observações que conduziram às conclusões deste trabalho.

4.3.1 Aceleração do Desempenho (*Speedup*)

O primeiro objetivo foi avaliar a variação de desempenho entre as versões adaptadas do *RST* que trabalham com o reuso localizado (exclusivamente dentro ou fora de laços) e a versão *RST* original, que emprega o reuso em todos os trechos do código, sempre que possível. A tabela B.1, no Anexo B, apresenta os resultados de *IPC* para as nove variações de arquitetura configuradas, ilustradas pelo gráfico da Figura 4.1.

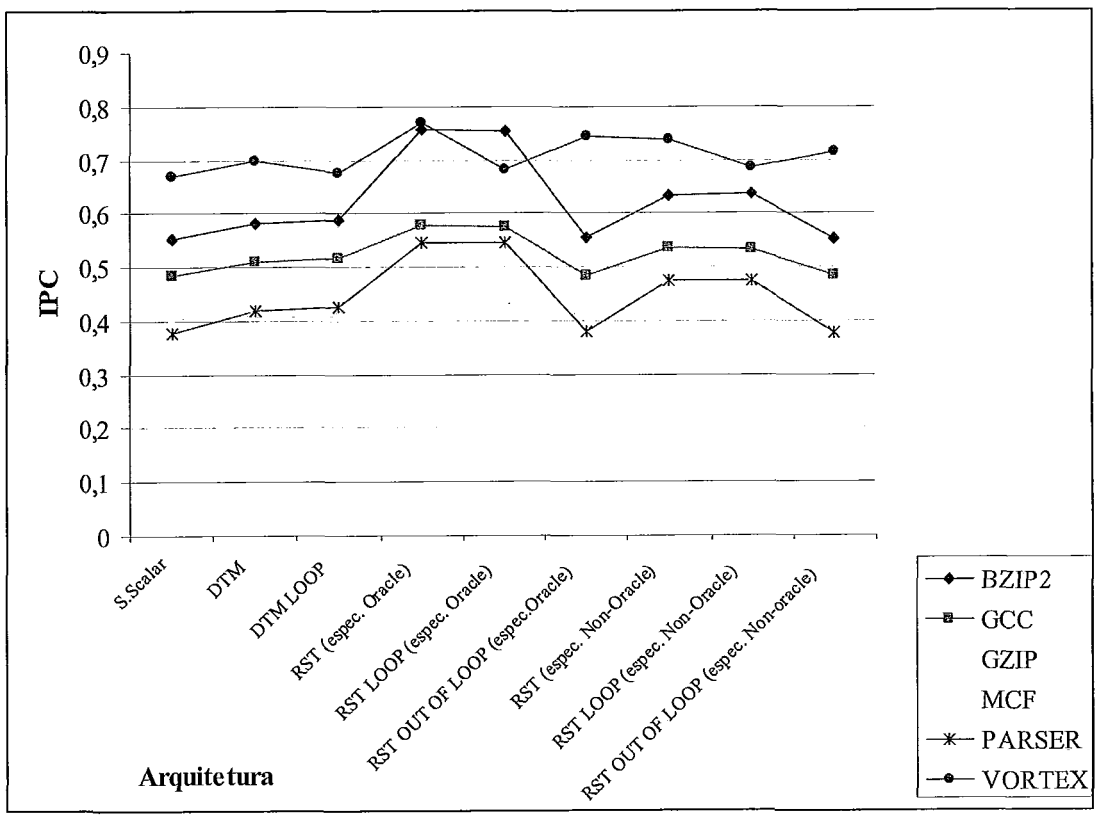


Figura 4.1 – IPC x Arquitetura

Dentre os resultados obtidos, destaca-se a proximidade da taxa de desempenho das versões *RST* e *RST LOOP* para a maioria dos programas de teste, além da inferioridade do resultado para a versão *RST OUT OF LOOP*, exceto no caso do benchmark *VORTEX*. Esta análise comparativa, como as demais apresentadas nesta seção, levará em conta os resultados das versões configuradas com o modo especulativo imperfeito (*"não-oráculo"*), quando este estiver presente, refletindo portanto uma arquitetura o mais realista possível.

O quadro comparativo da tabela 4.1 endossa a teoria motivadora deste trabalho, de que uma maioria significativa dos *traces* encontram-se dentro de estruturas de laços do código dos programas, retratada também pelo gráfico da Figura 4.2. Percebe-se que, quando empregadas as técnicas de reuso de forma localizada, apenas dentro dos laços detectados no código, o desempenho resultante aproxima-se e muito de uma abordagem tradicional, quando a abrangência é toda a extensão do código.

	RST LOOP sobre o RST					
IPC	***BZIP2***	***GCC***	***GZIP***	***MCF***	***PARSER***	***VORTEX***
	0,20%	**-0,20%**	**-1,20%**	**-0,20%**	**-0,08%**	**-7,10%**

Tabela 4.1 – *Speedup* da versão *RST LOOP* sobre a versão *RST*

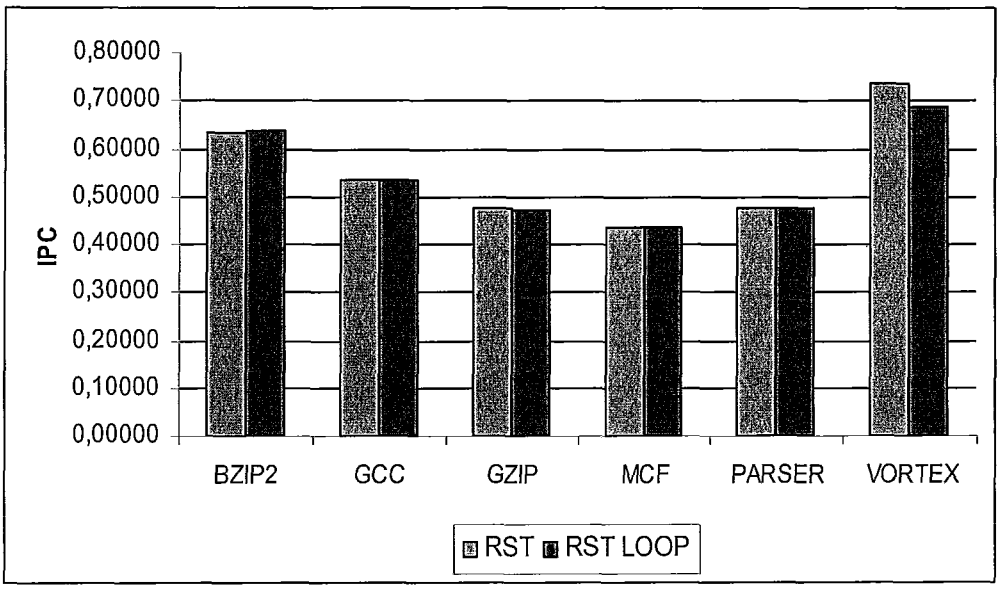


Figura 4.2 - *Speedup* da versão *RST LOOP* sobre a versão *RST*

Para averiguação dos resultados da versão *RST LOOP*, a Tabela 4.2 e o gráfico da Figura 4.3 exibem a aceleração de desempenho desta versão comparada ao *DTM*, confirmando-se a permanência do mesmo percentual de ganho aproximado que a versão *RST* já havia apresentado (PILLA; 2001, 2002, 2003a, 2003b), também representado pelo gráfico da figura 4.2. No caso do *VORTEX*, é apresentado também o *speedup* da versão *RST OUT OF LOOP* sobre o *DTM,* já que para este programa a versão que faz reuso exclusivamente fora dos laços apresentou melhores resultados.

	RST LOOP sobre o DTM						**Out Of Loop sobre DTM**
IPC	***BZIP2***	***GCC***	***GZIP***	***MCF***	***PARSER***	***VORTEX***	**VORTEX**
	9,20%	**4,40%**	**-2,20%**	**6,20%**	**13,46%**	**-2,00%**	**2,30%**

Tabela 4.2 – *Speedup* da versão *RST LOOP* sobre a versão *DTM*

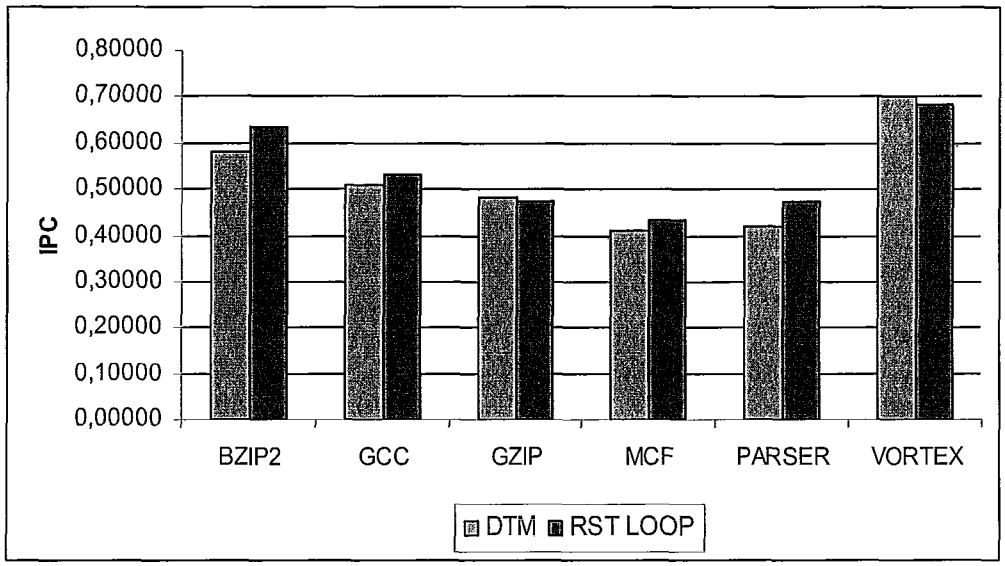


Figura 4.3 – *Speedup* da versão *RST LOOP* sobre a versão *DTM*

De forma similar à proximidade dos resultados encontrados para as versões *RST* e *RST LOOP*, constatou-se que o mesmo ocorre ao realizar-se um comparativo entre as versões *DTM* e *DTM LOOP* (Figura 4.4), cujos resultados refletem o que já havia sido constatado com o mecanismo *RST*, quanto à concentração dos *traces* dentro de laços, mas desta vez com a ausência de uma técnica de especulação.

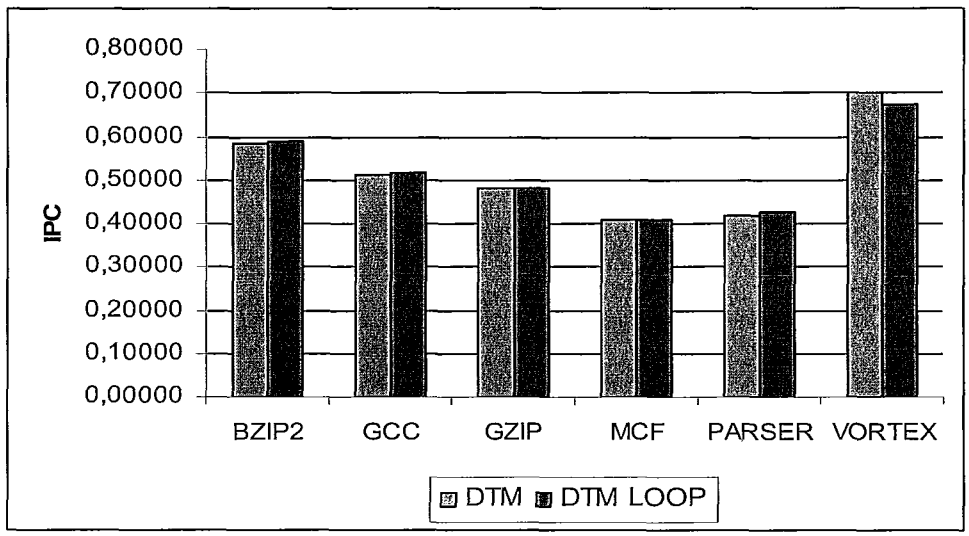


Figura 4.4 – *Speedup* da versão *DTM LOOP* sobre a versão *DTM*

Como mais uma evidência da forte correlação entre os *traces* reusáveis e os laços existentes no código dos programas, os resultados da versão *RST OUT OF LOOP*, que emprega o reuso localizado apenas em trechos do código externos a estruturas de laço, apresentaram um desempenho muito abaixo ao comparado com a versão *RST* original (Figura 4.5) e até com a versão *DTM* (Figura 4.6), exceto no caso do benchmark *VORTEX*, cujo código é uma exceção quanto ao comportamento observado nos demais programas.

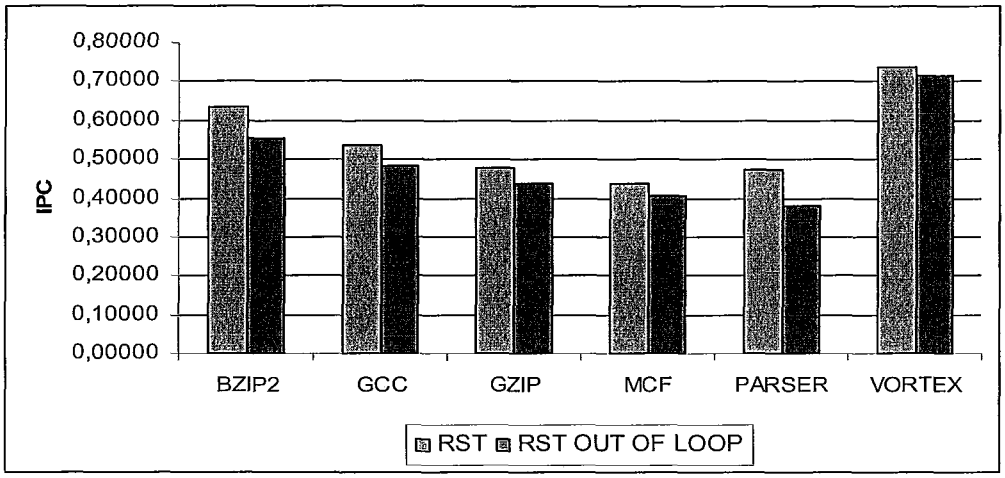


Figura 4.5 – *Speedup* da versão *RST OUT OF LOOP* sobre a versão *RST*

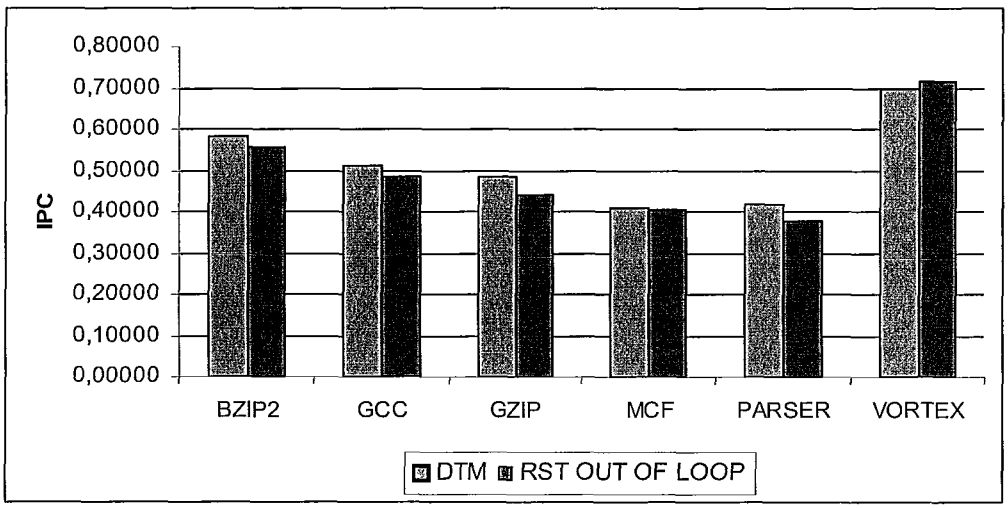


Figura 4.6 - *Speedup* da versão *RST OUT OF LOOP* sobre a versão *DTM*

4.3.2 Variação do Tamanho e Associatividade de *Memo_Table_T* x *Speedup*

Para continuação da análise comparativa entre as versões *RST e RST LOOP*, *DTM* e *DTM LOOP*, foi realizada mais uma série de experimentos tendo como foco o impacto no *speedup* da arquitetura sob a variação de dois fatores muito importantes da *Memo_Table_T*: o seu tamanho e sua associatividade.

O objetivo era constatar uma possível diferença de comportamento entre a versão original e a versão que faz reuso localizado apenas em laços. Inicialmente, manteve-se o fator de associatividade original da configuração, em 4 vias, variando-se apenas o tamanho da tabela, da faixa de 512 até 16 entradas.

Em seguida, repetiu-se a série de simulações com fatores de associatividade menores, primeiro de 2 vias e, por último, de apenas 1 via (mapeamento direto). Pelo perfil observado dos programas de teste quanto à variação do seu *IPC* nas várias arquiteturas simuladas até então, escolheu-se três destes particularmente para este levantamento, levando em consideração a diversidade de suas aplicações e o bom desempenho na execução da versão *RST LOOP*: *BZIP2*, *GCC* e *MCF*.

Adiante, as tabelas de resultados e seus respectivos gráficos representativos mostram que há uma redução significativa no desempenho do mecanismo *RST* (em modo especulativo "não-oráculo"), independentemente da adoção ou não da estratégia de reuso localizado em laços, em decorrência da diminuição do tamanho da tabela unificada de reuso de instruções e *traces* (Tabelas 4.3 à 4.5, Figuras 4.7 à 4.10).

GCC	Memo_Table_T (assoc. 4 vias)	512 entradas 128 conjuntos	256 entradas 64 conjuntos	128 entradas 32 conjuntos	64 entradas 16 conjuntos	32 entradas 8 conjuntos	16 entradas 4 conjuntos
RST LOOP	IPC	0,53340	0,52300	0,51390	0,50620	0,50050	0,49740
	TRACES	845.127.025	844.560.742	844.472.745	844.159.141	843.864.191	843.479.103
RST	IPC	0,53460	0,52360	0,51450	0,50650	0,50090	0,49760
	TRACES	861.251.449	860.770.338	860.395.716	860.108.174	859.670.211	859.267.260
DTM	IPC	0,51100	0,50990	0,50890	0,50780	0,50680	0,50610
	TRACES	2.352.762	2.845.736	3.683.710	4.958.787	6.764.368	9.675.162
DTM LOOP	IPC	0,51570	0,51370	0,51200	0,51040	0,50910	0,50800
	TRACES	2.424.210	3.079.377	4.224.524	5.729.478	7.881.297	11.491.466

Tabela 4.3 – Variação do *Speedup* do benchmark *GCC* sob variação do tamanho de Memo_Table_T

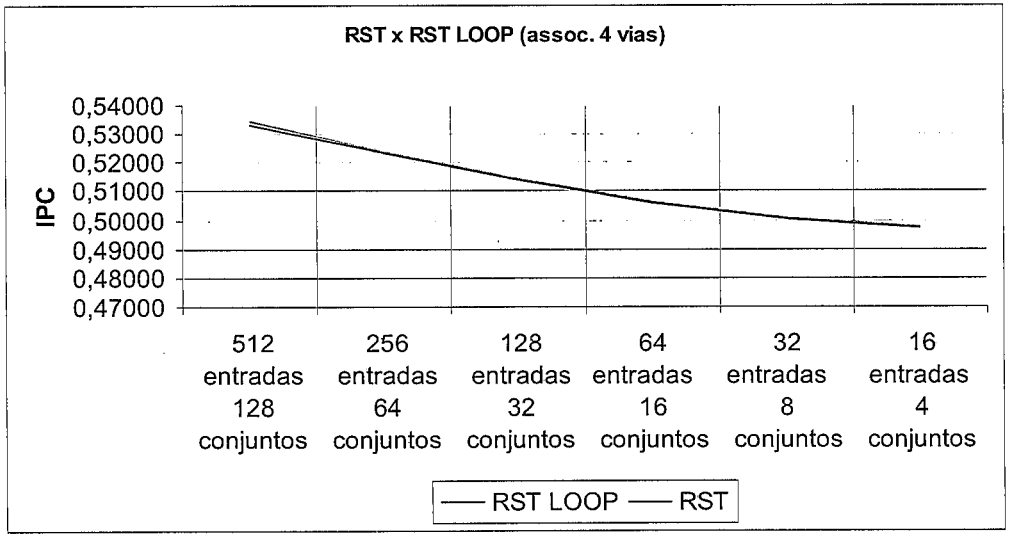


Figura 4.7 - Variação do *Speedup* do benchmark *GCC* sob variação do tamanho de Memo_Table_T

BZIP2	Memo_Table_T (assoc. 4 vias)	512 entradas 128 conjuntos	256 entradas 64 conjuntos	128 entradas 32 conjuntos	64 entradas 16 conjuntos	32 entradas 8 conjuntos	16 entradas 4 conjuntos
RST LOOP	IPC	0,63660	0,62450	0,60620	0,58790	0,56490	0,56020
	TRACES	766.372.752	766.375.453	766.174.391	766.068.239	765.879.514	765.633.626
RST	IPC	0,63510	0,62370	0,60590	0,59070	0,56680	0,56060
	TRACES	774.394.021	774.358.679	774.193.671	774.074.242	773.883.788	773.640.926
DTM	IPC	0,58290	0,58430	0,58300	0,58500	0,58250	0,58290
	TRACES	2.639.496	2.895.067	2.575.888	2.809.271	3.140.512	10.984.645
DTM LOOP	IPC	0,58780	0,58870	0,58740	0,58800	0,58510	0,58360
	TRACES	2.419.746	2.470.209	2.658.915	3.288.424	4.496.369	17.145.311

Tabela 4.4 – Variação do *Speedup* do benchmark *BZIP2* sob variação do tamanho de Memo_Table_T

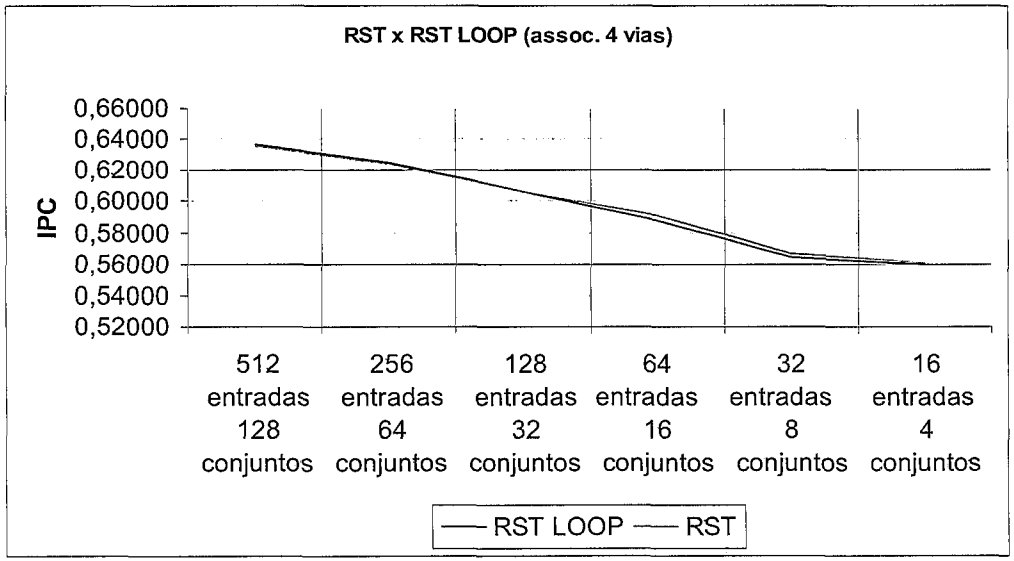


Figura 4.8 - Variação do *Speedup* do benchmark *BZIP2* sob variação do tamanho de Memo_Table_T

MCF	Memo_Table_T (assoc. 4 vias)	512 entradas 128 conjuntos	256 entradas 64 conjuntos	128 entradas 32 conjuntos	64 entradas 16 conjuntos	32 entradas 8 conjuntos	16 entradas 4 conjuntos
RST LOOP	IPC	0,43530	0,43360	0,43310	0,43240	0,43060	0,42750
	TRACES	641.306.645	641.337.679	641.403.692	641.343.925	641.344.014	641.256.491
RST	IPC	0,43640	0,43400	0,43360	0,43310	0,43110	0,42760
	TRACES	694.649.911	694.610.881	694.489.254	694.319.220	694.221.170	694.089.137
DTM	IPC	0,40990	0,41000	0,41000	0,41000	0,41000	0,40980
	TRACES	1.254.828	367.824	500.473	567.541	858.254	2.228.089
DTM LOOP	IPC	0,40920	0,40920	0,40940	0,40920	0,40940	0,40940
	TRACES	166.634	166.634	166.821	171.833	189.870	207.492

Tabela 4.5 – Variação do *Speedup* do benchmark *MCF* sob variação do tamanho de Memo_Table_T

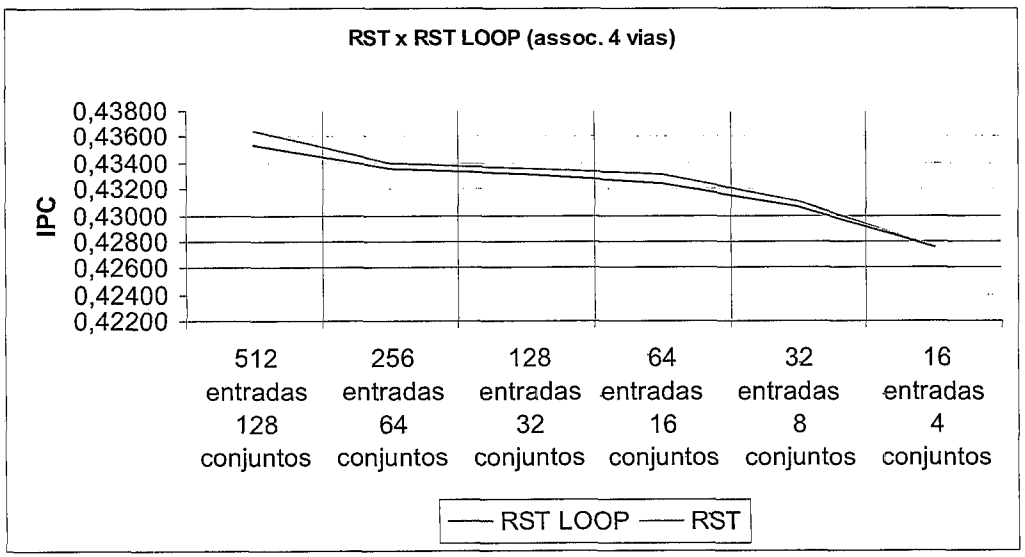


Figura 4.9 - Variação do *Speedup* do benchmark *MCF* sob variação do tamanho de Memo_Table_T

Para o mecanismo DTM, constatou-se que não houve uma redução significativa do *speedup* devido à redução do tamanho de *Memo_Table_T*, o que pode ser explicado pelo fato desta tabela ser usada apenas para o armazenamento de *traces* redundantes, havendo uma tabela em separado para o armazenamento das instruções (*Memo_Table_G)*, que acaba por ter um volume de dados armazenados bem maior.

No caso da configuração *RST,* como já citado, optou-se por trabalhar com uma tabela unificada para o armazenamento de *traces* e instruções redundantes, devido aos melhores resultados obtidos com esta configuração em trabalhos anteriores (PILLA; 2001, 2002, 2003a, 2003b).

Diferentemente do comportamento observado quanto à variação do *IPC* sob a redução do tamanho da tabela de reuso, este impacto não é tão forte para o *RST* quando se varia apenas a associatividade da tabela, havendo uma redução mínima do *speedup* à medida que se reduz o fator de associatividade (Tabelas 4.6 à 4.8, Figuras 4.10 à 4.12) .

Quanto à variação na associatividade, constata-se que para tabelas menores, o mapeamento direto torna-se a melhor opção para o *RST.* O tamanho da tabela, a partir da qual a associatividade de *n* vias pode ser substituída pelo mapeamento direto, varia para cada programa de teste. No caso do *GCC*, tabelas de 32 entradas ou menores, enquanto que para o *BZIP2* tabelas com 128 entradas ou menores já oferecem um melhor desempenho com a associatividade de única via.

GCC	Memo_Table_T (assoc. 2 vias)	512 entradas 256 conjuntos	256 entradas 128 conjuntos	128 entradas 64 conjuntos	64 entradas 32 conjuntos	32 entradas 16 conjuntos	16 entradas 8 conjuntos
RST LOOP	IPC	0,52790	0,51980	0,51200	0,50550	0,50070	0,49750
	TRACES	845.006.892	844.752.981	844.588.248	844.217.654	843.771.478	843.384.766
RST	IPC	0,52920	0,52090	0,51250	0,50590	0,50100	0,49780
	TRACES	861.255.839	860.776.463	860.393.890	860.061.383	859.646.473	859.277.500
GCC	Memo_Table_T (assoc. 1 via)	512 entradas 512 conjuntos	256 entradas 256 conjuntos	128 entradas 128 conjuntos	64 entradas 64 conjuntos	32 entradas 32 conjuntos	16 entradas 16 conjuntos
RST LOOP	IPC	0,52420	0,51770	0,51140	0,50530	0,50070	0,49770
	TRACES	845.043.974	844.629.704	844.433.201	843.972.217	843.795.763	843.477.010
RST	IPC	0,52580	0,51890	0,51220	0,50590	0,50100	0,49780
	TRACES	861.219.376	860.731.177	860.315.573	859.963.936	859.622.735	859.314.347

Tabela 4.6 – Variação do *Speedup* do benchmark *GCC* sob variação da associatividade e do tamanho de Memo_Table_T

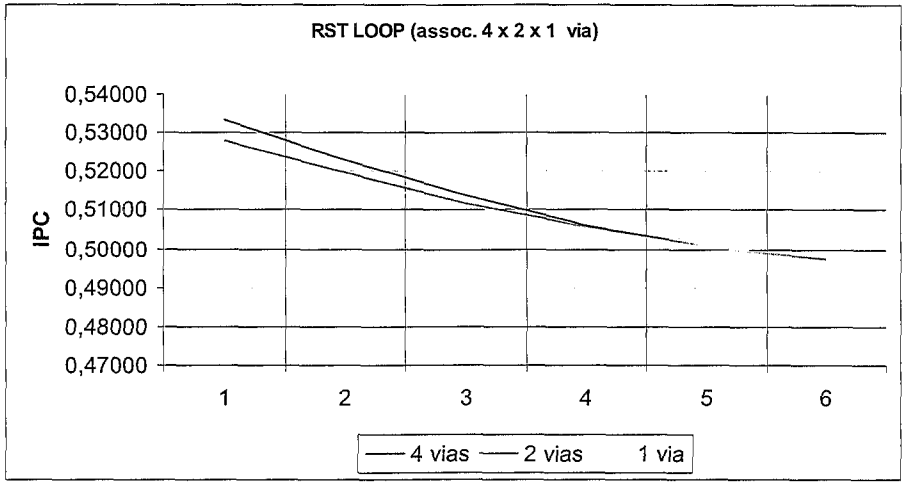


Figura 4.10 - Variação do *Speedup* do benchmark *GCC* sob variação da associatividade de Memo_Table_T

BZIP2	Memo_Table_T (assoc. 2 vias)	512 entradas 256 conjuntos	256 entradas 128 conjuntos	128 entradas 64 conjuntos	64 entradas 32 conjuntos	32 entradas 16 conjuntos	16 entradas 8 conjuntos
RST LOOP	IPC	0,63220	0,61960	0,61030	0,58600	0,57770	0,56220
	TRACES	766.228.522	766.262.459	766.172.312	765.987.969	765.835.674	765.689.563
RST	IPC	0,63220	0,61990	0,60990	0,58710	0,57820	0,56250
	TRACES	774.252.633	774.338.514	774.163.261	774.002.167	773.846.059	773.694.524
BZIP2	**Memo_Table_T (assoc. 1 via)**	**512 entradas 512 conjuntos**	**256 entradas 256 conjuntos**	**128 entradas 128 conjuntos**	**64 entradas 64 conjuntos**	**32 entradas 32 conjuntos**	**16 entradas 16 conjuntos**
RST LOOP	IPC	0,63280	0,62430	0,61300	0,60280	0,58360	0,56350
	TRACES	766.192.021	766.195.972	766.144.681	766.062.280	766.024.268	765.607.075
RST	IPC	0,63110	0,62390	0,61330	0,60380	0,58510	0,56350
	TRACES	774.248.816	774.247.062	774.192.639	773.989.903	774.083.926	773.617.848

Tabela 4.7 – Variação do *Speedup* do benchmark *BZIP2* sob variação da associatividade e do tamanho de Memo_Table_T

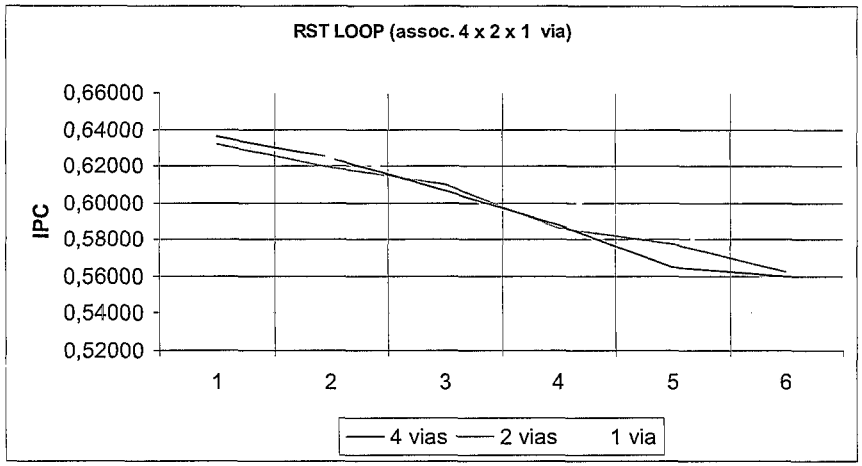


Figura 4.11 - Variação do *Speedup* do benchmark *BZIP2* sob variação da associatividade de Memo_Table_T

MCF	Memo_Table_T (assoc. 2 vias)	512 entradas 256 conjuntos	256 entradas 128 conjuntos	128 entradas 64 conjuntos	64 entradas 32 conjuntos	32 entradas 16 conjuntos	16 entradas 8 conjuntos
RST LOOP	IPC	0,43730	0,43480	0,43350	0,43280	0,43230	0,42410
	TRACES	641.312.990	641.327.461	641.342.648	641.373.655	641.337.164	641.277.604
RST	IPC	0,43750	0,43500	0,43350	0,43330	0,43290	0,42460
	TRACES	694.568.873	694.512.234	694.478.998	694.340.561	694.204.003	694.079.048
MCF	**Memo_Table_T (assoc. 1 via)**	**512 entradas 512 conjuntos**	**256 entradas 256 conjuntos**	**128 entradas 128 conjuntos**	**64 entradas 64 conjuntos**	**32 entradas 32 conjuntos**	**16 entradas 16 conjuntos**
RST LOOP	IPC	0,43390	0,43440	0,43380	0,43310	0,43140	0,42800
	TRACES	641.325.184	641.359.843	641.306.581	641.315.338	641.372.591	641.272.641
RST	IPC	0,43450	0,43490	0,43360	0,43280	0,43140	0,42850
	TRACES	694.488.871	694.488.025	694.414.166	694.414.608	694.260.345	694.099.826

Tabela 4.8 – Variação do *Speedup* do benchmark *MCF* sob variação da associatividade e do tamanho de Memo_Table_T

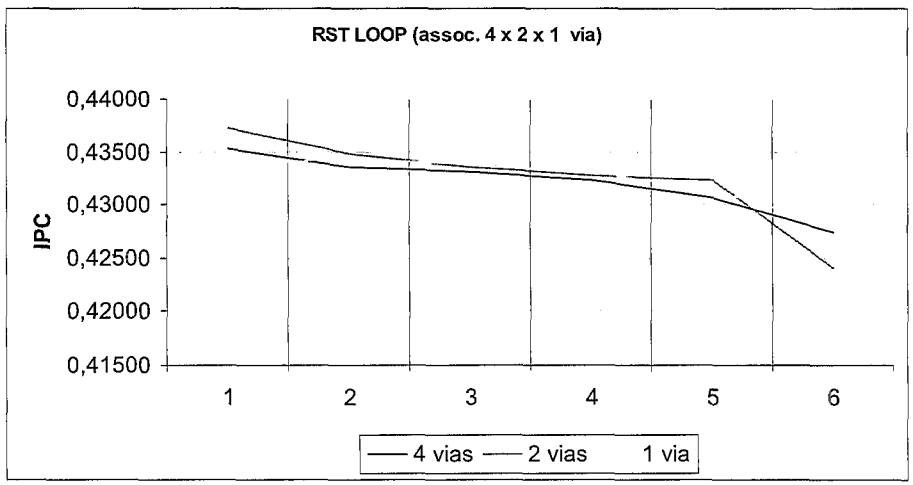


Figura 4.12 - Variação do *Speedup* do benchmark *MCF* sob variação da associatividade de Memo_Table_T

4.3.3 Variação nos acessos à *Memo_Table_T* e às Estruturas de Memória

Além do desempenho (*IPC*), outro fator observado atentamente foi o número de *traces* capturados por cada configuração de arquitetura, que traduz na realidade o número de acessos feitos à tabela de reuso. Nesta seção, é feita uma análise sobre a variação do número destes acessos quanto às configurações testadas, assim como dos acessos feitos também às estruturas de memória de cada arquitetura.

O objetivo de um estudo desta natureza é encontrar pontos de redução no número de acessos a certas estruturas, que possibilitem a diminuição do seu tamanho ou redistribuição deste entre as demais estruturas da arquitetura, visando encontrar um balanceamento ideal para os *tradeoffs* de desempenho e de consumo de energia, de maior importância ainda para as aplicações que têm como alvo sistemas embarcados.

Para este propósito, fez-se um levantamento do número total de acessos feitos à *Memo_Table_T*, à cache de instruções de primeiro nível IL1, à cache de dados de primeiro nível DL1 e à tabela do Preditor de Desvios, durante todo o período das simulações executadas para os *benchmarks*, em cada arquitetura. Na tabela C1, no Anexo C, há o levantamento completo destes dados, que são pontualmente analisados a seguir.

O primeiro ponto interessante observado é a redução do número de acessos à *Memo_Table_T* na versão *RST LOOP* em relação ao *RST* original, variando na faixa de 1% a 12%. A quantidade de acessos às estruturas de cache e à tabela do Preditor de Desvios também sofre uma redução na maioria dos casos, mas o comportamento é particular a cada benchmark, como visto na Tabela 4.9. Para o programa de teste *VORTEX* sempre também se leva em consideração, como citado anteriormente, os resultados da versão *RST OUT OF LOOP* (Tabela 4.21).

	RST LOOP x RST					
	BZIP2	**GCC**	**GZIP**	**MCF**	**PARSER**	**VORTEX**
IPC	**0,20%**	**-0,20%**	**-1,20%**	**-0,20%**	**-0,08%**	**-7,10%**
Captured *Traces*	-1,00%	-2,00%	-12,00%	-7,60%	-1,60%	-78,50%
DL1 accesses	-0,10%	0,04%	0,03%	-0,20%	-0,40%	1,10%
IL1 accesses	0,30%	0,07%	-0,30%	-0,20%	-0,30%	-0,70%
BPRED Table Lookups	-0,07%	0,10%	-0,50%	0,20%	-0,20%	-1,10%

Tabela 4.9 – Ganho Percentual de acessos da versão *RST LOOP* sobre a versão *RST*

	RST OUT OF LOOP x RST
	VORTEX
IPC	**-3,00%**
Captured *Traces*	-21,53%
DL1 accesses	0,26%
IL1 accesses	-0,21%
BPRED Table Lookups	-0,24%

Tabela 4.10 – Ganho Percentual de acessos da versão *RST OUT OF LOOP* sobre a versão *RST,* para o benchmark *VORTEX*

Além da comparação entre as versões que usam o mecanismo *RST*, fez-se o mesmo para uma análise diferencial entre as versões *DTM* e *RST LOOP*, de forma que seja possível vislumbrar o aumento de *speedup* gerado pela segunda técnica e o aumento no número de acessos a maioria das estruturas que esta ocasiona (Tabela 4.11). Neste caso, o número de acessos à *Memo_Table_T* não foi incluído, pois as arquiteturas estão configuradas de forma diferente, uma vez que a versão *DTM* utiliza duas tabelas

distintas para o reuso de instruções e *traces*, enquanto que a versão *RST LOOP* usa uma tabela unificada.

	RST LOOP x DTM					
	BZIP2	**GCC**	**GZIP**	**MCF**	**PARSER**	**VORTEX**
IPC	**9,20%**	**4,40%**	**-2,20%**	**6,20%**	**13,46%**	**-2,00%**
DL1 accesses	-0,79%	-0,25%	-1,23%	2,00%	0,72%	0,31%
IL1 accesses	4,66%	3,78%	4,13%	0,16%	2,88%	1,33%
BPRED Table Lookups	5,02%	3,81%	6,75%	-0,44%	3,17%	1,92%

Tabela 4.11 - Ganho Percentual de acessos da versão *RST LOOP* sobre a versão *DTM*

	RST OUT OF LOOP x DTM
	VORTEX
IPC	**2,30%**
DL1 accesses	-0,57%
IL1 accesses	1,87%
BPRED Table Lookups	2,77%

Tabela 4.12 - Ganho Percentual de acessos da versão *RST OUT OF LOOP* sobre a versão *DTM*

No comparativo entre as versões *RST* e *RST LOOP*, percebe-se que, exceto para o programa de teste *PARSER*, todos os demais programas apresentam redução no número de acessos a algumas estruturas e acréscimo no acesso a outras, não havendo um padrão. A única taxa que sofre redução significativa para todos os programas é a de acessos à *Memo_Table_T*.

Para a avaliação sobre uma eventual redução geral no número de acessos feitos a estas estruturas, este segundo levantamento tratou o número absoluto de acessos realizados a cada estrutura, sendo possível posteriormente comparar o total de acessos ocorrido nas simulações com as duas versões.

Pelas Tabelas 4.13 à 4.15, pode-se afirmar que a versão *RST LOOP*, além de manter o mesmo nível de *speedup* proporcionado pela versão *RST*, oferece uma redução geral no número de acessos feito às estruturas de memória da arquitetura, na média em torno de 0,63%, desconsiderando o programa *VORTEX* e o *GZIP*, este último pelo fato de ser o único a apresentar um acréscimo de 0,10% no número total de acessos.

A redução significativa no número de acessos à *Memo_Table_T* não tem seu ganho sobrepujado por um eventual aumento no número de acessos às estruturas de memória de primeiro nível e à tabela do Preditor de Desvios, o que comprova a sua propensão à redução no consumo de energia.

		RST	RST LOOP
BZIP2	*TRACES*	774.394.021	766.372.752
	DL1 accesses	556.419.100	555.834.425
	IL1 accesses	4.712.146.814	4.728.211.189
	Bpred accesses	805.940.507	805.371.659
	Total	6.848.900.442	6.855.790.025
	Ganho Percent.		0,10%

		RST	RST LOOP
GCC	*TRACES*	861.251.449	845.127.025
	DL1 accesses	797.800.979	798.152.940
	IL1 accesses	4.684.157.705	4.687.638.389
	Bpred accesses	936.112.005	937.169.209
	Total	7.279.322.138	7.268.087.563
	Ganho Percent.		-0,15%

Tabela 4.13 – Ganho no número total de acessos da versão *RST LOOP* sobre a versão *RST* para os programas *BZIP* e *GCC*

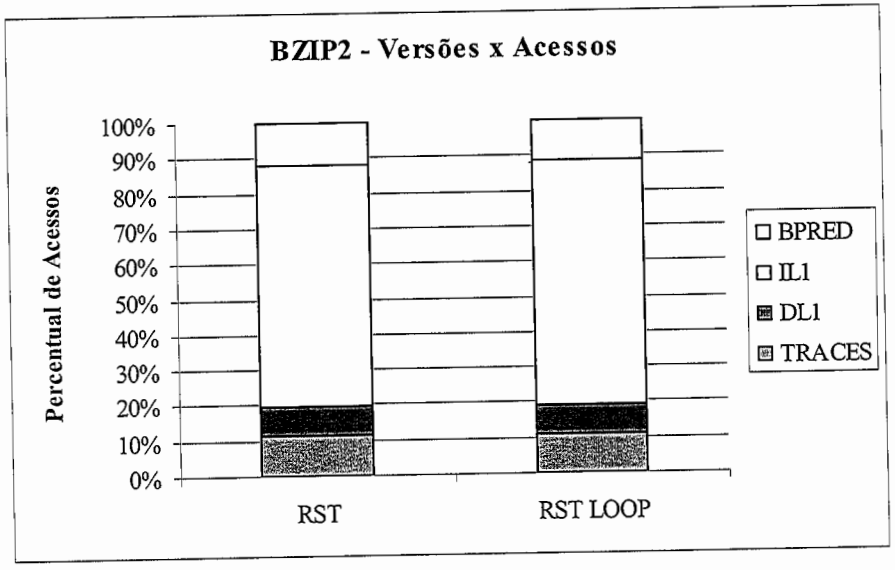


Figura 4.13 – Totalização dos Acessos no BZIP2

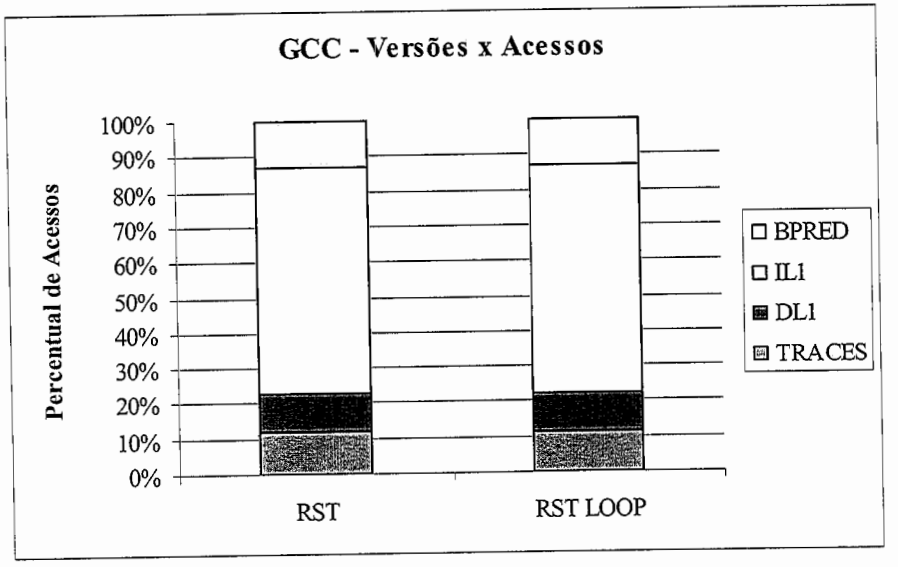


Figura 4.14 – Totalização dos Acessos no GCC

		RST	RST LOOP
MCF	*TRACES*	694.649.911	641.306.645
	DL1 accesses	681.954.271	680.378.171
	IL1 accesses	6.885.616.529	6.867.221.772
	Bpred accesses	1.921.437.264	1.925.790.554
	Total	10.183.657.975	10.114.697.142
	Ganho Percent.		-0,68%

		RST	RST LOOP
GZIP	*TRACES*	722.896.191	636.308.870
	DL1 accesses	533.858.977	534.007.772
	IL1 accesses	7.111.287.441	7.087.148.271
	Bpred accesses	1.236.524.689	1.229.778.383
	Total	9.604.567.298	9.487.243.296
	Ganho Percent.		-1,22%

Tabela 4.14 – Ganho no número total de acessos da versão *RST LOOP* sobre a versão *RST* para os programas *MCF* e *GZIP*

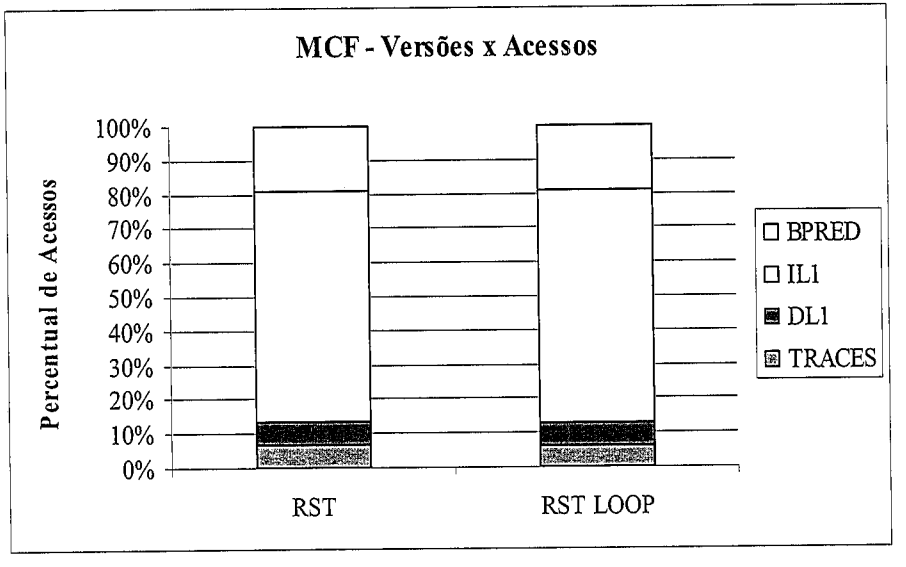


Figura 4.15 – Totalização dos Acessos no MCF

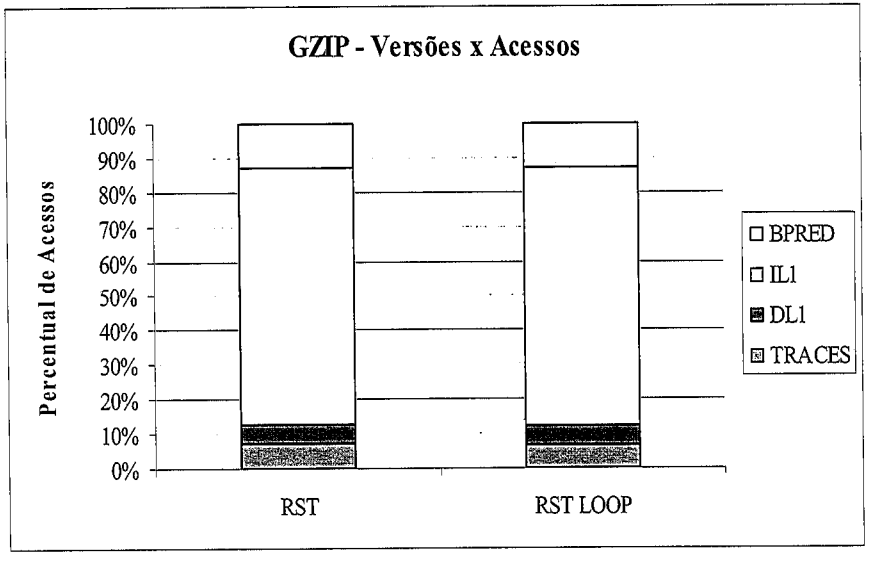


Figura 4.16 – Totalização dos Acessos no GZIP

		RST	RST LOOP
PARSER	*TRACES*	752.628.926	740.023.684
	DL1 accesses	768.656.538	765.093.174
	IL1 accesses	5.682.626.883	5.661.671.455
	Bpred accesses	1.297.352.647	1.293.956.743
	Total	8.501.264.994	8.460.745.056
	Ganho Percent.		-0,48%

		RST	RST LOOP
VORTEX	*TRACES*	954.216.580	204.585.700
	DL1 accesses	829.415.762	838.972.849
	IL1 accesses	3.355.139.660	3.330.004.320
	Bpred accesses	549.050.995	542.942.481
	Total	5.687.822.997	4.916.505.350
	Ganho Percent.		-13,56%

Tabela 4.15 – Ganho no número total de acessos da versão *RST LOOP* sobre a versão *RST* para os programas *PARSER* e *VORTEX*

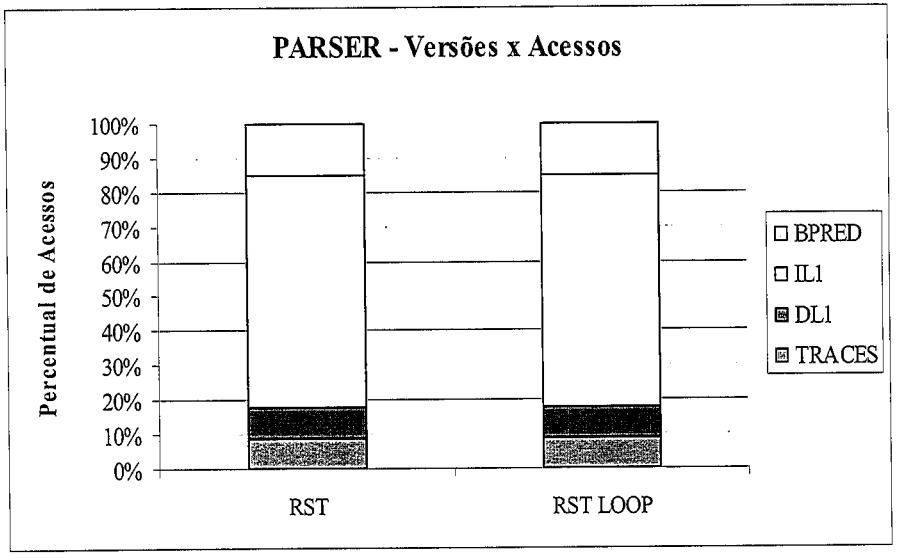


Figura 4.17 – Totalização dos Acessos no PARSER

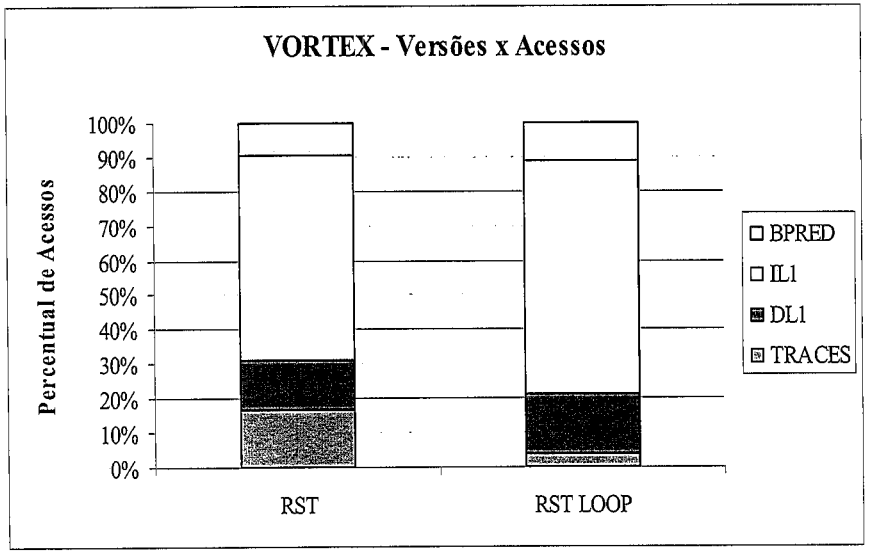


Figura 4.18 – Totalização dos Acessos no VORTEX

4.3.4 *Tradeoff* entre a *Cache L1* e a *Memo_Table_T*

Tendo em vista a redução no número de *traces* capturados e no número total de acessos feitos às estruturas de armazenamento da arquitetura proporcionados pela versão *RST LOOP*, os experimentos seguintes realizados tiveram como foco avaliar qual destas estruturas tem maior impacto no *speedup* quando tem seu tamanho significativamente aumentado. Mais especificamente, esta análise restringiu-se a três estruturas: a *cache* de dados de primeiro nível *L1*, a *cache* de instruções de primeiro nível *L1* e a tabela de reuso *Memo_Tabel_T*.

Para esta avaliação, iniciou-se as simulações com uma arquitetura-base composta de duas *caches L1* de 32KB cada, uma para instruções (*IL1*) e outra para dados (*DL1*). A tabela *Memo_Table_T* composta por 8 conjuntos e 32 entradas. A escolha por uma tabela de reuso pequena é justificada pelo propósito desta seção, que visa prover resultados que direcionem para qual destas estruturas deve-se dedicar o investimento disponível, de forma a incrementar o desempenho da arquitetura.

O quadro da Tabela D.1, no Anexo D, exibe a coleta dos resultados das simulações realizadas para todo o subconjunto de benchmarks do *SPECint2000* testado neste trabalho, desde a arquitetura-base inicial, passando por diversas variações de sua configuração quanto ao tamanho das *caches L1* e da tabela de reuso.

É possível perceber que a redução do tamanho das *caches L1*, de dados ou de instruções, não resulta em uma perda de desempenho, gerando apenas um crescimento no número de acessos à cache. Dobrando-se o tamanho das *caches L1*, o desempenho também permaneceu inalterada para a maioria dos programas, com um crescimento médio de 0,02%. Considerando o caso em houvesse a possibilidade de aumento do *budget* de memória da arquitetura, a comparação sobre os resultados da

Tabela E.1, no Anexo E, busca responder em qual estrutura este investimento estaria mais bem empregado.

Como ilustrado pelo gráfico da Figura 4.19, os ganhos percentuais de desempenho indicam que, para esta configuração de arquitetura, o aumento no *budget* de memória deveria ser direcionado ao incremento do tamanho da tabela de reuso, ao invés das *caches L1*. Enquanto que a configuração na qual se duplicou o tamanho das *caches L1* não surtiu em qualquer melhora no desempenho, a outra arquitetura, em que se manteve o tamanho original das *caches* de primeiro nível e duplicou-se o tamanho da tabela de reuso, apresentou um ganho médio de 1,7% no desempenho, sob a penalidade de um acréscimo de 0,88% no número de acessos feitos às estruturas de memória.

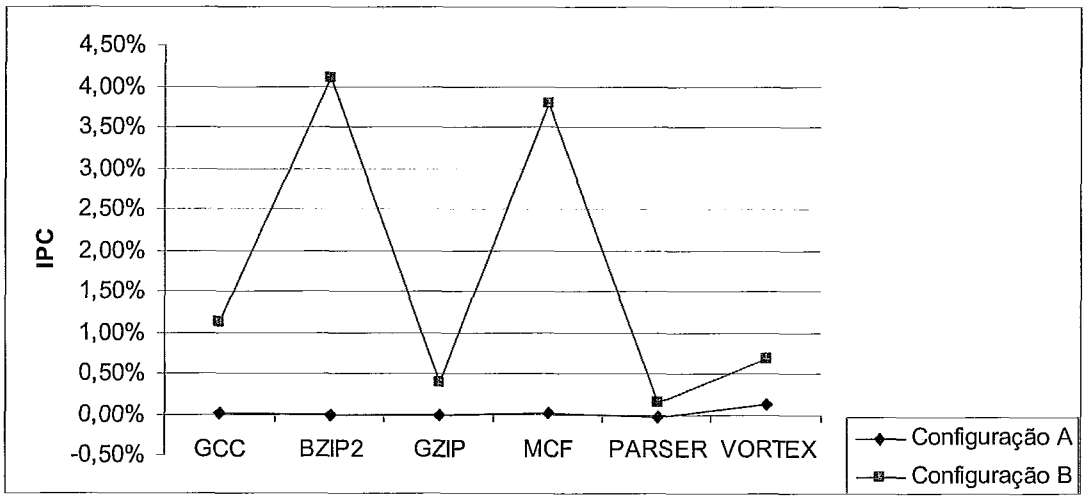


Figura 4.19 – Ganho IPC x Configuração da *Cache L1* e da *Memo_Table_T*

Confirguração A – *Memo_Table_T* com 32 entradas e 8 conjuntos (associativa de 4 vias) e *Caches L1* de Instrução e Dados com 64KB de tamanho cada.

Configuração B – *Memo_Table_T* com 64 entradas e 16 conjuntos (associativa de 4 vias) e *Caches L1* de Instrução e Dados com 32KB de tamanho cada.

CAPÍTULO 5 – Conclusões

Neste trabalho de Dissertação, buscou-se avaliar inicialmente qual o padrão dos *paths* que antecedem os *traces* redundantes capturados no código dos programas para que, em seguida, fossem avaliadas as possibilidades de se fazer um *prefetch* destes *traces* de modo a obter um maior ganho no desempenho de uma arquitetura superescalar com o mecanismo *RST*. O levantamento do perfil destes *paths* indicaram os laços como os grandes concentradores de *traces* redundantes e, a partir deste indício, foram adaptadas duas versões de arquitetura que empregam o reuso de *traces* e instruções pelo mecanismo *RST* de forma localizada: o reuso apenas no interior das estruturas de laços, ou o reuso apenas externamente a estas.

Utilizando um subconjunto de programas de teste do pacote *SPECint2000*, foram realizadas simulações com estas versões modificadas, que usam os laços como fronteiras para ativação e desativação do mecanismo de reuso, e com a versão original. O uso da técnica de especulação e o nível da sua taxa de acertos também foram fatores variados durante estes experimentos. Endossando a teoria de que a maioria dos *traces* redundantes encontram-se em laços, constatou-se pelos resultados das simulações que a versão *sim-rst-loop*, empregando reuso localizado apenas internamente aos laços, manteve o mesmo nível de *speedup* que a versão *sim-rst* original apresentara sobre o *DTM*. Outras análises sobre a gama de resultados destas simulações levaram às seguintes conclusões:

Há uma redução do número de *traces* capturados na versão *sim-rst-loop*, comparada ao *RST* original, entre 1 e 12%, dependendo do benchmark, mantendo-se o mesmo nível de desempenho, além de uma redução, na média em torno de 0,63%, no número total de acessos feitos às estruturas de memória de primeiro nível. Esta

manutenção no número total de acessos às estruturas foi decorrente da compensação observada entre a redução nos acessos à Tabela de Reuso e o acréscimo dos acessos às *Caches L1* e à Tabela do Preditor de Desvios.

• Há uma redução significativa no desempenho do mecanismo *RST* (em modo especulativo "não-oráculo"), independentemente da adoção ou não da estratégia de reuso localizado em laços, em decorrência da diminuição do tamanho da tabela unificada de reuso de instruções e *traces*. Para o mecanismo DTM, constatou-se que não houve uma redução significativa do *speedup* devido à redução do tamanho de *Memo_Table_T*, o que pode ser explicado pelo fato desta tabela ser usada apenas para o armazenamento de *traces* redundantes, existindo uma tabela em separado para o armazenamento das instruções (*Memo_Table_G*). A grande diferença no número de acessos à Memo_Table_T nas versões *DTM* e *RST* indicam que a maior parte de acessos é feita para o armazenamento/reuso de uma instrução isolada;

• Quanto à associatividade, a variação da mesma tem baixo impacto sobre o desempenho do *RST*, independentemente da estratégia de reuso. Percebe-se que para tabelas menores o mapeamento direto é mais eficiente, obviamente. A decisão sobre o tamanho da tabela a partir do qual se torna mais interessante utilizar o mapeamento direto, ao invés de uma associatividade de *n* vias, varia para cada programa de teste. Por exemplo, no caso do BZIP2 e do MCF, para uma tabela com até 256 entradas o mapeamento direto já é mais interessante, enquanto que para o GCC apenas para tabelas com 32 ou menos entradas;

• A última análise fez referência ao possível *tradeoff* entre o investimento nas *caches L1* ou na tabela unificada de reuso. Foram realizadas simulações variando-se o tamanho das *caches L1*, a partir de uma configuração-base padrão com duas *caches L1* de 32KB, uma para instruções e outra para dados, e uma tabela de reuso com 32

entradas, de tamanho aproximado aos das *caches*. Considerando-se a possibilidade de um aumento na ordem de 100% no *budget* de memória, obteve-se um melhor desempenho (*speedup* de 1,7% contra 0,2%), nesta configuração, aumentando-se o tamanho da tabela de reuso, ao invés de aumentar-se o tamanho das *caches L1*, sob pouca penalidade, em torno de 0,88%, quanto ao acréscimo no número de acessos às estruturas;

- Além da manutenção no número total de acessos às estruturas, constatou-se que o reuso aplicado apenas internamente aos laços não resulta em uma economia significativa de energia para a arquitetura, como mostrado pela Tabela F.1, no Anexo F, mas possibilita uma eventual redução do tamanho da tabela unificada de reuso devido ao menor número de *traces* capturados. Para o cálculo do consumo de energia no acesso a cada estrutura (Tabela F.3, no Anexo F), foi utilizada a ferramenta CACTI 4.2 (*Hewlett-Packard Development Company.* Disponível em http://quid.hpl.hp.com:9081/cacti/. Acesso em 07/04/2008). As configurações das estruturas de memória, passadas como parâmetros para a execução desta ferramenta, estão descritas na Tabela F.2, no Anexo F.

Referências Bibliográficas

Da COSTA, A.T., FRANÇA, F.M.G. *The Reuse Potencial of Trace Memoization.* Technical Report ES-498/99, COPPE/UFRJ, May 1999.

Da COSTA, A. T. *Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador.* Abril, 2001. Tese. (Doutorado em Ciência da Computação) — COPPE–UFRJ.

Da COSTA, A.T., FRANÇA, F.M.G., CHAVES, E.M.F. *The Dynamic Trace Memoization Reuse Technique.* In Proc. of International Conference on Parallel Architectures and Compilation Techniques, pp.92-99, October 2000.

GONZALEZ, A., TUBELLA, J., MOLINA, C. *Trace-Level Reuse.* In Proc. of International Conference on Parallel Processing, pp.30-37, September 1999.

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

HEWLETT-PACKARD Development Company. CACTI Project, 2008. Disponível em http://quid.hpl.hp.com:9081/cacti/. Acesso em 07/04/2008).

HUANG, J., LILJA, D.J. *Exploiting Basic Block Value Locality with Block Reuse.* In Proc. of 5th International Symposium on High-Performance Computer Architecture, pp. 106-114, January 1999.

HUANG, J., LILJA, D.J. *Exploring Sub-Block Value Reuse for SuperScalar Processors.* In Proc.of International Conference on Parallel Architectures and Compilation Techniques, pp.100-107, October 2000.

KOUSHIRO, T., SATO, T., ARITA, I. *A Trace-Level Value Predictor for Contrail Processors.* SIGARCH Computer Architecture News 31(3): 42-47 (2003).

LIAO, C.-H.; SHIEH, J.-J. *Exploiting Speculative Value Reuse Using Value Prediction.* In: ASIA-PACIFIC CONFERENCE ON COMPUTER SYSTEMS ARCHITECTURE, 7., 2002. Australian Computer Society: Inc.. p.101–108, 2002.

MICHIE, D. *Memo Functions and Machine Learning. Nature* v. 1, n.218, pp. 19-22, April 1968.

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F.M. G, *et al. Predicting Trace Inputs with Dynamic Trace Memoization: determining speedup upper bounds.* IEEE TCCA Newsletter, [S.l.], October 2001.

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G.; *et al. Speculative Trace Reuse.* Porto Alegre: [s.n.], 2002. Tech Report. (RP–320).

PILLA, M. L.; NAVAUX, P. O. A.; FRANÇA, F. M. G, *et al. Improving Performance through Speculative Trace Reuse.* In: CADERNOS DE INFORMÁTICA –

PROCESSAMENTO PARALELO E DISTRIBUÍDO NA INFORMÁTICA/UFRGS, 2003, Porto Alegre. Instituto de Informática – UFRGS, 2003. v.3, n.1, p.139 – 144. Presented in the Workshop of the Parallel and Distribuited Processing Group, 2003, Porto Alegre.

PILLA, M. L. *RST: Reuse through Speculation on Traces*. Abril, 2004. Tese. (Doutorado em Ciência da Computação) — Instituto de Informática – UFRGS.

PILLA, M.L., NAVAUX, P.O.A., da COSTA, A.T., *et al. Reuse Through Speculation on Traces for Deeply Pipelined SuperScalars Processors*. Technical Report 345, II-UFRGS, 2004.

PILLA, M.L., NAVAUX, P.O.A., da COSTA, A.T., *et al. The Limits of Speculative Trace Reuse on Deeply Pipelined Processors*. In Proc. of 15th Symposium on Computer Architectures and High Performance Computing, p.36-44, October 2003.

PILLA, M.L., NAVAUX, P.O.A., FRANÇA, F.M.G, *et al. A Speculative Trace Reuse Architecture with Reduced Hardware Requirements*. In Proc. of 18th International Symposium on Computer Architecture and High Performance Computing, 2006.

PILLA, M.L., NAVAUX, P.O.A., FRANÇA, F.M.G, *et al. Limits for a Feasible Speculative Trace Reuse Implementation*. Innovative Computing and Applications, Vol. 1, Nº 1, 2007.

SODANI, A. *Dynamic Instruction Reuse.* 2000. Tese (Doutorado em Ciência da Computação) — University of Wisconsin, Madison.

SODANI, A., SOHI, G.S. *Dynamic Instruction Reuse.* In Proc. of 24th Annual International Symposium on Computer Architecture, pages 194-205, July 1997.

SODANI, A., SOHI, G.S. *Understanding Differences Between Value Prediction and Instruction Reuse.* In Proc. of 31th Annual International Symposium on Microarchitecture, pp. 205-215, December 1998.

TSUMARA, T., SUZUKI, I., IKEUCHI, Y., *et al. Design and Evaluation of na Auto-Memoization Processor.* Parallel and Distributed Computing and Networks 2007.

ANEXO A

Configuração do Pipeline		
Parâmetro	**Valor**	**Descrição**
-pipe:fetch	4	número de sub-estágios do estágio de Busca.
-pipe:decode	4	número de sub-estágios do estágio de Decodificação.
-pipe:dispatch	2	número de sub-estágios do estágio de Despacho.
-pipe:issue	5	número de sub-estágios do estágio de Emissão (*issue*).
-pipe:exec	1	número de sub-estágios do estágio de Execução.
-pipe:writeback	2	número de sub-estágios do estágio de Writeback.
-pipe:commit	1	número de sub-estágios do estágio de Entrega (*commit*).
-decode:width	4	largura de banda do estágio de Decodificação. Número de instruções que este estágio pode entregar em um único ciclo de clock.
-issue:width	4	largura de banda do estágio de Emissão. Número de instruções que este estágio pode entregar em um único ciclo de clock.
-commit:width	4	largura de banda do estágio de Entrega. Número de instruções que este estágio pode entregar em um único ciclo de clock.
-ruu:size	128	Tamanho da RUU em número de estações de reserva.
-lsq:size	64	Tamanho da fila de Load/Stores (LSQ queue) em número de estações de reserva.

Tabela A.1 – Configuração do Pipeline

Configuração do Preditor de Desvios		
Parâmetro	**Valor**	**Descrição**
-bpred	2lev	tipo de Preditor de Desvios. Preditor de Desvios em 2 níveis.
-bpred:2lev	1 8192 13 1	Configuração do Preditor de Desvios em 2 níveis (<l1size> <l2size> <hist_size> <xor>).

Tabela A.2 – Configuração do Preditor de Desvios

Configuração das Estruturas de Cache		
(<nome>:<número de conjuntos>:<tamanho do bloco>:<associatividade) Na configuração escolhida, há duas caches de primeiro nível distintas, uma para instruções e outra apenas para dados. As caches de segundo e terceiro níveis são unificadas para o armazenamento de blocos de instruções e dados.		
Parâmetro	**Valor**	**Descrição**
-cache:dl1	dl1:128:64:4:l	Configuração da cache de dados L1.
-cache:dl2	ul2:256:256:8:l	Configuração da cache de dados L2.
-cache:dl3	ul3:1024:256:8:l	Configuração da cache de dados L3.
-cache:il1	il1:64:128:4:l	Configuração da cache de instruções L1.
-cache:il2	dl2	Configuração da cache de instruções L2.
-cache:il3	dl3	Configuração da cache de instruções L3.

Tabela A.3 – Configuração das Estruturas de Cache

Recursos Funcionais		
Parâmetro	**Valor**	**Descrição**
-mem:width	16	largura em bytes do barramento de acesso à memória principal.
-res:ialu	2	número de ALU's disponíveis para operação com inteiros.
-res:imult	1	número de multiplicadores/divisores de inteiros disponíveis.
-res:memport	2	número de portas do sistema de memória disponíveis para acesso pela CPU.
-res:fpalu	1	número de ALU's disponíveis para operação com números de ponto-flutuante.
-res:fpmult	1	número de multiplicadores/divisores disponíveis para operação com números de ponto-flutuante.

Tabela A.4 – Recursos Funcionais

Amostra		
Parâmetro	**Valor**	**Descrição**
-max:commit	1500000000	Número máximo de instruções a serem executadas e entregues.
-fastfwd	500000000	Número de instruções a serem executadas em modo funcional, sem a simulação do tempo gasto nos estágios do pipeline e sem a coleta de estatísticas pelo simulador.

Tabela A.5 – Amostra

Configuração do SimpleScalar sem Reuso		
Parâmetro	**Valor**	**Descrição**
-dtm:notrace	true	não realizar o reuso de traços.
-dtm:noinst	true	não realizar o reuso de instruções.
-vp:oracle	false	usar previsão de valores perfeita (modo "oráculo").
-vp:predict	false	usar previsão de valores em modo "não-oráculo".

Tabela A.6 – Configuração do SimpleScalar sem Reuso

Configuração do *DTM / DTM LOOP*		
Parâmetro	**Valor**	**Descrição**
-dtm:allow_reusable	false	permitir a inclusão de instruções reusáveis não redundantes nos traces.
-dtm:notrace	false	não realizar o reuso de traços.
-dtm:nobpred	false	Caso o mecanismo DTM esteja ativado, o mesmo não atualiza a tabela do Preditor de Desvios. Com o valor false, a atualização será feita.
-dtm:noinst	false	não realizar o reuso de instruções.
-dtm:inputscope	2	tamanho do escopo de entrada dos traces capturados
-dtm:outputscope	1	tamanho do escopo de saída dos traces capturados
-dtm:min_trace_size	2	tamanho mínimo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-dtm:max_trace_size	64	tamanho máximo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-MT:T	512	Número de entradas em Memo_Table_T.
-MT:I	512	Número de entradas em Memo_Table_G.
-rst:memo_t_assoc	4	Associatividade de Memo_Table_T.
-rst:memo_t_sets	128	Número de conjuntos em Memo_Table_T.
-rst:memo_g_assoc	4	Associatividade de Memo_Table_G.
-rst:memo_g_sets	128	Número de conjuntos em Memo_Table_G.
-vp:oracle	false	usar previsão de valores perfeita (modo "oráculo").
-vp:predict	false	usar previsão de valores em modo "não-oráculo".

Tabela A.7 – Configuração das Arquiteturas *DTM* e *DTM LOOP*

Configuração das versões *RST / RST LOOP / RST OUT OF LOOP com Especulação Non-Oracle*		
Parâmetro	**Valor**	**Descrição**
-dtm:allow_reusable	true	permitir a inclusão de instruções reusáveis não redundantes nos traces.
-dtm:notrace	false	não realizar o reuso de traços.
-dtm:nobpred	false	Caso o mecanismo DTM esteja ativado, o mesmo não atualiza a tabela do Preditor de Desvios. Com o valor false, a atualização será feita.
-dtm:noinst	true	não realizar o reuso de instruções.
-dtm:inputscope	2	tamanho do escopo de entrada dos traces capturados
-dtm:outputscope	1	tamanho do escopo de saída dos traces capturados
-dtm:min_trace_size	1	tamanho mínimo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-dtm:max_trace_size	64	tamanho máximo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-MT:T	512	Número de entradas em Memo_Table_T.
-MT:I	512	Número de entradas em Memo_Table_G.
-rst:memo_t_assoc	4	Associatividade de Memo_Table_T.
-rst:memo_t_sets	128	Número de conjuntos em Memo_Table_T.
-rst:memo_g_assoc	4	Associatividade de Memo_Table_G.
-rst:memo_g_sets	128	Número de conjuntos em Memo_Table_G.
-vp:oracle	false	usar previsão de valores perfeita (modo "oráculo").
-vp:predict	true	usar previsão de valores em modo "não-oráculo".
-vp:trace_size_threshold	1	tamanho mínimo de um trace para previsão.
-confid	counter	política do mecanismo de confiança do preditor.
-confid:counter	4096 3 3 2 1 1	configuração do mecanismo de confiança - counter (<table size> <threshold> <saturation> <penalty> <increment> <initial>).
-confid:percent	90	taxa percentual de acerto do mecanismo de confiança (<hit rate>).

Tabela A.8 – Configuração das versões ***RST / RST LOOP / RST OUT OF LOOP com Especulação Non-Oracle***

Configuração das versões *RST / RST LOOP / RST OUT OF LOOP com Especulação Oracle*		
Parâmetro	**Valor**	**Descrição**
-dtm:allow_reusable	true	permitir a inclusão de instruções reusáveis não redundantes nos traces.
-dtm:notrace	false	não realizar o reuso de traços.
-dtm:nobpred	false	Caso o mecanismo DTM esteja ativado, o mesmo não atualiza a tabela do Preditor de Desvios. Com o valor false, a atualização será feita.
-dtm:noinst	true	não realizar o reuso de instruções.
-dtm:inputscope	2	tamanho do escopo de entrada dos traces capturados
-dtm:outputscope	1	tamanho do escopo de saída dos traces capturados
-dtm:min_trace_size	1	tamanho mínimo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-dtm:max_trace_size	64	tamanho máximo, em número de instruções, de um trace a ser armanzenado em Memo_Table_T.
-MT:T	512	Número de entradas em Memo_Table_T.
-MT:I	512	Número de entradas em Memo_Table_G.
-rst:memo_t_assoc	4	Associatividade de Memo_Table_T.
-rst:memo_t_sets	128	Número de conjuntos em Memo_Table_T.
-rst:memo_g_assoc	4	Associatividade de Memo_Table_G.
-rst:memo_g_sets	128	Número de conjuntos em Memo_Table_G.
-vp:oracle	true	usar previsão de valores perfeita (modo "oráculo").
-vp:predict	false	usar previsão de valores em modo "não-oráculo".
-vp:trace_size_threshold	1	tamanho mínimo de um trace para previsão.

Tabela A.9 – Configuração das versões RST / RST LOOP / RST OUT OF LOOP com Especulação Oracle

ANEXO B

		S.Scalar Original (sem reuso)	**DTM** (RST sem especulação)	**DTM LOOP** (RST LOOP sem especulação)	**RST** com especulação Oracle	**RST LOOP** com especulação Oracle	**RST OUT OF LOOP** com especulação Oracle	**RST** com especulação Non-Oracle	**RST LOOP** com especulação Non-Oracle	**RST OUT OF LOOP** com especulação Non-oracle
BZIP2	**IPC**	0,55100	0,58290	0,58780	0,75760	0,75330	0,55560	0,63510	0,63660	0,55340
GCC	**IPC**	0,48410	0,51100	0,51570	0,57790	0,57540	0,48570	0,53460	0,53340	0,48480
GZIP	**IPC**	0,43720	0,48310	0,48150	0,48580	0,47910	0,44190	0,47830	0,47230	0,43990
MCF	**IPC**	0,40310	0,40990	0,40920	0,43760	0,43060	0,40940	0,43640	0,43530	0,40510
PARSER	**IPC**	0,37810	0,41880	0,42530	0,54680	0,54560	0,37960	0,47560	0,47520	0,37830
VORTEX	**IPC**	0,66950	0,70000	0,67610	0,76950	0,68220	0,74370	0,73820	0,68540	0,71620

Tabela B.1 – Resultados de Desempenho (*IPC*) das Arquiteturas

ANEXO C

		S.Scalar	DTM	DTM LOOP	RST (espec.Oracle)	RST LOOP (espec. Oracle)	RST OUT OF LOOP (espec. Oracle)	RST (espec. Non-Oracle)	RST LOOP (espec. Non-Oracle)	RST OUT OF LOOP (espec. Non-oracle)
BZIP2	IPC	0,55100	0,58290	0,58780	0,75760	0,75330	0,55560	0,63510	0,63660	0,55340
	TRACES	0	2.639.496	2.419.746	773.160.398	765.204.253	7.990.569	774.394.021	766.372.752	8.005.054
	DL1 accesses	551.827.036	560.268.927	558.128.881	507.458.190	509.175.270	550.099.800	556.419.100	555.834.425	551.957.615
	IL1 accesses	4.544.898.883	4.517.776.962	4.463.625.604	3.900.348.375	3.926.939.093	4.716.028.334	4.712.146.814	4.728.211.189	4.613.393.434
	Bpred accesses	772.442.689	766.864.332	757.267.728	662.162.386	665.740.580	804.327.103	805.940.507	805.371.659	785.065.819
GCC	IPC	0,48410	0,51100	0,51570	0,57790	0,57540	0,48570	0,53460	0,53340	0,48480
	TRACES	0	2.352.762	2.424.210	858.528.958	843.003.429	15.756.913	861.251.449	845.127.025	15.853.180
	DL1 accesses	800.610.209	800.113.817	797.696.456	769.938.737	770.153.565	800.278.040	797.800.979	798.152.940	800.824.982
	IL1 accesses	4.683.054.754	4.516.989.438	4.501.057.976	4.170.519.186	4.185.810.404	4.671.102.444	4.684.157.705	4.687.638.389	4.686.955.232
	Bpred accesses	938.430.370	902.768.604	899.999.339	832.290.773	835.606.072	936.135.344	936.112.005	937.169.209	938.275.214
GZIP	IPC	0,43720	0,48310	0,48150	0,48580	0,47910	0,44190	0,47830	0,47230	0,43990
	TRACES	0	1.237.514	3.045.791	721.049.765	633.258.141	110.293.916	722.896.191	636.308.870	110.869.110
	DL1 accesses	519.722.561	540.581.650	540.119.983	518.573.986	518.510.237	521.208.233	533.858.977	534.007.772	523.089.347
	IL1 accesses	7.128.248.939	6.794.753.031	6.811.408.129	6.837.803.787	6.854.277.426	7.178.217.802	7.111.287.441	7.087.148.271	7.250.513.316
	Bpred accesses	1.215.396.229	1.146.755.454	1.145.068.141	1.174.927.330	1.175.702.946	1.221.731.775	1.236.524.689	1.229.778.383	1.235.388.271
MCF	IPC	0,40310	0,40990	0,40920	0,43760	0,43060	0,40940	0,43640	0,43530	0,40510
	TRACES	0	1.254.828	166.634	693.529.874	640.774.609	51.637.391	694.649.911	641.306.645	52.147.211
	DL1 accesses	669.298.233	666.804.615	676.395.147	680.671.263	681.498.982	667.510.597	681.954.271	680.378.171	671.336.227
	IL1 accesses	6.774.195.581	6.855.921.716	6.931.436.751	6.501.048.289	6.629.738.956	6.644.027.456	6.885.616.529	6.867.221.772	6.792.866.529
	Bpred accesses	1.917.941.411	1.934.201.421	1.936.206.795	1.841.413.231	1.875.253.910	1.885.454.040	1.921.437.264	1.925.790.554	1.915.980.283
PARSER	IPC	0,37810	0,41880	0,42530	0,54680	0,54560	0,37960	0,47560	0,47520	0,37830
	TRACES	0	8.156.392	4.895.998	749.163.046	736.674.710	11.488.774	752.628.926	740.023.684	11.468.311
	DL1 accesses	753.935.336	759.554.270	759.968.516	723.597.238	720.940.482	753.300.296	768.656.538	765.093.174	754.955.718
	IL1 accesses	5.747.784.179	5.498.452.517	5.476.190.373	4.811.554.222	4.817.551.307	5.729.007.579	5.682.626.883	5.661.671.455	5.764.654.874
	Bpred accesses	1.313.085.758	1.252.945.175	1.245.904.782	1.102.968.390	1.104.698.497	1.308.532.969	1.297.352.647	1.293.956.743	1.317.214.278
VORTEX	IPC	0,66950	0,70000	0,67610	0,76950	0,68220	0,74370	0,73820	0,68540	0,71620
	TRACES	0	1.830.374	408.632	952.304.319	204.301.313	747.363.361	954.216.580	204.585.700	748.742.075
	DL1 accesses	841.325.606	836.330.787	839.415.781	816.657.073	838.214.332	819.361.274	829.415.762	838.972.849	831.580.991
	IL1 accesses	3.332.877.388	3.285.676.744	3.309.469.596	3.032.398.219	3.291.573.796	3.099.717.434	3.355.139.660	3.330.004.320	3.348.225.303
	Bpred accesses	541.776.209	532.514.955	537.055.496	492.227.527	536.468.546	504.649.097	549.050.995	542.942.481	547.714.012

Tabela C.1 – Quadro com totais de acessos às estruturas *Memo_Table_T*, cache *DL1*, cache *IL1* e Tabela de Previsão de Desvios

ANEXO D

RST LOOP (espec. Non-Oracle)	Memo_Table_T (assoc. 4 vias)	32 entradas 8 conjuntos	32 entradas 8 conjuntos	32 entradas 8 conjuntos	32 entradas 8 conjuntos	64 entradas 16 conjuntos	32 entradas 8 conjuntos	64 entradas 16 conjuntos
	Cache L1	dl1:128:64:4 il1:64:128:4 (original)	dl1:64:64:4 il1:64:128:4	dl1:128:64:4 il1:32:128:4	dl1:64:64:4 il1:32:128:4	dl1:64:64:4 il1:32:128:4	dl1:256:64:4 il1:128:128:4	dl1:128:64:4 il1:64:128:4 (original)
GCC	IPC	0,50050	0,50050	0,50050	0,50050	0,50610	0,50060	0,50620
	TRACES	843.864.191	843.750.544	843.862.970	843.854.906	844.156.113	843.929.830	844.159.141
	IL1 accesses	4.721.177.376	4.724.671.118	4.723.543.886	4.722.340.682	4.726.333.812	4.724.165.891	4.727.227.674
	DL1 accesses	806.104.256	806.433.289	806.447.603	806.263.143	805.589.502	806.573.495	805.726.398
BZIP2	IPC	0,56490	0,56600	0,56530	0,56490	0,58820	0,56480	0,58790
	TRACES	765.879.514	765.879.409	765.879.434	765.877.418	766.072.454	765.878.640	766.068.239
	IL1 accesses	4.798.578.155	4.771.806.983	4.804.063.352	4.745.597.547	4.945.043.355	4.774.483.485	4.941.458.230
	DL1 accesses	557.314.800	558.373.431	557.395.516	557.331.046	560.692.464	557.253.621	560.424.635
GZIP	IPC	0,44110	0,44110	0,44110	0,44110	0,44410	0,44110	0,44410
	TRACES	609.669.428	609.659.088	609.634.276	609.659.088	610.072.398	609.677.593	610.067.938
	IL1 accesses	7.221.636.970	7.221.602.692	7.220.235.842	7.221.602.395	7.272.735.237	7.221.604.899	7.274.147.601
	DL1 accesses	525.772.308	525.812.651	525.757.799	525.812.651	528.137.651	525.779.272	528.156.793
MCF	IPC	0,43060	0,43060	0,43060	0,43060	0,43240	0,43070	0,43240
	TRACES	641.344.014	641.343.666	641.344.014	641.343.666	641.346.093	641.348.356	641.343.925
	IL1 accesses	6.860.863.796	6.864.021.075	6.860.863.796	6.864.021.103	6.865.702.072	6.858.006.069	6.865.407.732
	DL1 accesses	679.567.964	679.535.689	679.567.964	679.535.689	679.678.818	679.635.519	679.503.262
PARSER	IPC	0,38300	0,38290	0,38290	0,38290	0,39730	0,38290	0,39740
	TRACES	738.095.946	738.099.557	738.094.329	738.087.478	738.621.501	738.082.943	738.607.962
	IL1 accesses	5.889.503.761	5.890.502.980	5.892.706.065	5.891.927.639	5.983.170.973	5.890.802.073	5.983.204.981
	DL1 accesses	765.902.756	765.614.382	765.969.839	765.708.316	768.502.995	765.712.170	768.675.520
VORTEX	IPC	0,67880	0,67890	0,67850	0,67830	0,68020	0,67960	0,68060
	TRACES	204.643.743	204.645.043	204.640.218	204.640.610	204.486.756	204.637.514	204.488.023
	IL1 accesses	3.335.581.158	3.340.691.105	3.321.817.469	3.330.247.052	3.334.235.301	3.328.191.071	3.348.119.042
	DL1 accesses	840.152.037	840.189.748	838.367.754	838.953.522	837.434.855	840.194.952	838.968.715

OBS: Configuração da Cache: <nome>:<número de conjuntos>:<tamanho do bloco>:<associatividade>

Tabela D.1 – *Speedup* sob a variação do tamanho das *caches L1* e da *Memo_Table_T*

ANEXO E

RST LOOP (espec. Non-Oracle)	Memo_Table_T (assoc. 4 vias)	32 entradas 8 conjuntos	32 entradas 8 conjuntos	64 entradas 16 conjuntos
	Cache L1	dl1:128:64:4 il1:64:128:4	dl1:256:64:4 il1:128:128:4	dl1:128:64:4 il1:64:128:4
GCC	IPC	0,50050	0,50060	0,50620
	TRACES	843.864.191	843.929.830	844.159.141
	IL1 accesses	4.721.177.376	4.724.165.891	4.727.227.674
	DL1 accesses	806.104.256	806.573.495	805.726.398
	Ganho IPC		0,02%	1,12%
	Ganho Acessos		0,06%	0,04%
BZIP2	IPC	0,56490	0,56480	0,58790
	TRACES	765.879.514	765.878.640	766.068.239
	IL1 accesses	4.798.578.155	4.774.483.485	4.941.458.230
	DL1 accesses	557.314.800	557.253.621	560.424.635
	Ganho IPC		-0,02%	4,09%
	Ganho Acessos		-0,39%	2,79%
GZIP	IPC	0,44110	0,44110	0,44410
	TRACES	609.669.428	609.677.593	610.067.938
	IL1 accesses	7.221.636.970	7.221.604.899	7.274.147.601
	DL1 accesses	525.772.308	525.779.272	528.156.793
	Ganho IPC		0,00%	0,68%
	Ganho Acessos		0,00%	0,66%
MCF	IPC	0,43060	0,43070	0,43240
	TRACES	641.344.014	641.348.356	641.343.925
	IL1 accesses	6.860.863.796	6.858.006.069	6.865.407.732
	DL1 accesses	679.567.964	679.635.519	679.503.262
	Ganho IPC		0,02%	0,39%
	Ganho Acessos		-0,03%	0,09%
PARSER	IPC	0,38300	0,38290	0,39740
	TRACES	738.095.946	738.082.943	738.607.962
	IL1 accesses	5.889.503.761	5.890.802.073	5.983.204.981
	DL1 accesses	765.902.756	765.712.170	768.675.520
	Ganho IPC		-0,03%	3,79%
	Ganho Acessos		0,01%	1,30%
VORTEX	IPC	0,67880	0,67960	0,68060
	TRACES	204.643.743	204.637.514	204.488.023
	IL1 accesses	3.335.581.158	3.328.191.071	3.348.119.042
	DL1 accesses	840.152.037	840.194.952	838.968.715
	Ganho IPC		0,12%	0,15%
	Ganho Acessos		-0,17%	0,42%
	Ganho Médio IPC		0,02%	1,70%
	Ganho Médio Acessos		-0,09%	0,88%

Tabela E.1 – *Speedup* e número de acessos sob a variação em dobro do tamanho das estruturas

ANEXO F

		RST (espec. Non-Oracle)	Gasto de Energia (nJ)	RST LOOP (espec. Non-Oracle)	Gasto de Energia (nJ)
	IPC	0,6351		0,6366	
	TRACES	774.394.021	35.882.952,13	766.372.752	35.511.272,07
	DL1 accesses	556.419.100	44.939.836,38	555.834.425	44.892.614,43
	IL1 accesses	4.712.146.814	649.231.274,76	4.728.211.189	651.444.596,01
	Bpred accesses	805.940.507	7.016.512,86	805.371.659	7.011.560,47
BZIP2	TOTAL	6.848.900.442	737.070.576	6.855.790.025	738.860.043
	IPC	0,5346		0,5334	
	TRACES	861.251.449	39.907.648,66	845.127.025	39.160.494
	DL1 accesses	797.800.979	64.435.325	798.152.940	64.463.751
	IL1 accesses	4.684.157.705	645.374.985	4.687.638.389	645.854.547
	Bpred accesses	936.112.005	8.149.785	937.169.209	8.158.989
GCC	TOTAL	7.279.322.138	757.867.743	7.268.087.563	757.637.782
	IPC	0,4783		0,4723	
	TRACES	722.896.191	33.496.707	636.308.870	29.484.526
	DL1 accesses	533.858.977	43.117.742	534.007.772	43.129.759
	IL1 accesses	7.111.287.441	979.780.638	7.087.148.271	976.454.785
	Bpred accesses	1.236.524.689	10.765.176	1.229.778.383	10.706.443
GZIP	TOTAL	9.604.567.298	1.067.160.262	9.487.243.296	1.059.775.514
	IPC	0,4364		0,4353	
	TRACES	694.649.911	32.187.864	641.306.645	29.716.107
	DL1 accesses	681.954.271	55.078.831	680.378.171	54.951.535
	IL1 accesses	6.885.616.529	948.688.098	6.867.221.772	946.153.700
	Bpred accesses	1.921.437.264	16.728.020	1.925.790.554	16.765.920
MCF	TOTAL	10.183.657.975	1.052.682.813	10.114.697.142	1.047.587.263
	IPC	0,4756		0,4752	
	TRACES	752.628.926	34.874.427	740.023.684	34.290.340
	DL1 accesses	768.656.538	62.081.440	765.093.174	61.793.641
	IL1 accesses	5.682.626.883	782.942.307	5.661.671.455	780.055.105
	Bpred accesses	1.297.352.647	11.294.744	1.293.956.743	11.265.179
PARSER	TOTAL	8.501.264.994	891.192.917	8.460.745.056	887.404.265
	Média comparativa (%)		**-0,49%**		**-0,28%**

Tabela F.1 – Comparativo entre as versões *RST* e *RST-LOOP* quanto ao gasto de energia nos acessos às estruturas de memória.

CONFIGURAÇÕES
Memo_Table_T (assoc. 4 vias) - 512 entradas de 24 bytes em 128 conjuntos.
Cache dl1 - dl1:128:64:4 - <nome>:<número de conjuntos>:<tamanho do bloco>:<associatividade>
Cache il1 - il1:64:128:4 - <nome>:<número de conjuntos>:<tamanho do bloco>:<associatividade>
Bpred_Table - 1 8192 13 1 - Preditor de Desvios em 2 níveis (<l1size> <l2size> <hist_size> <xor>).

Tabela F.2 – Configurações das estruturas de memória usadas para o comparativo de consumo de energia.

Estrutura	Gasto de Energia no Acesso (nJ)
Memo_Table_T	0,04633681453120
DL1	0,08076616418210
IL1	0,13777823577900
BPRED	0,00870599355095

Tabela F.3 – Gasto de energia no acesso a cada estrutura.