COPPE/UFRJ

SENSIBILIDADE DO MECANISMO DE REUSO DE TRAÇOS AOS SUBCONJUNTOS DE INSTRUÇÕES

Sheila de Oliveira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas da Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Ciências em Engenharia de Sistemas da Computação.

Orientador: Felipe Maia Galvão França

Rio de Janeiro

Março de 2009

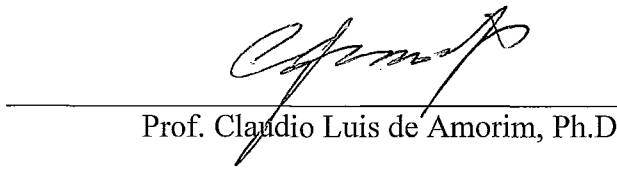
SENSIBILIDADE DO MECANISMO DE REUSO DE TRAÇOS AOS SUBCONJUNTOS DE INSTRUÇÕES

Sheila de Oliveira

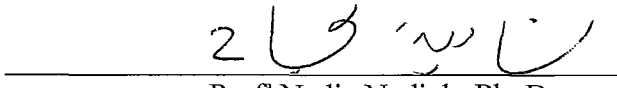
DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS DA COMPUTAÇÃO.

Aprovada por:

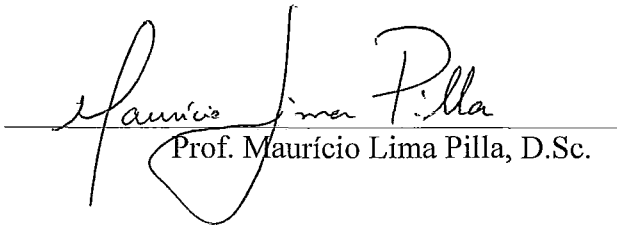
Prof. Felipe Maia Galvão França, Ph.D.



Prof. Claudio Luis de Amorim, Ph.D.



Profª Nadia Nedjah, Ph. D.



Prof. Maurício Lima Pilla, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2009

Oliveira, Sheila de

Sensibilidade do Mecanismo de Reuso de Traços aos subconjuntos de instruções/Sheila de Oliveira. – Rio de Janeiro: UFRJ/COPPE, 2009.

XII, 82 p.: il.; 29,7cm.

Orientador: Felipe Maia Galvão França

Dissertação (mestrado) – UFRJ / COPPE / Programa de Engenharia de Sistemas da Computação, 2009.

Referências Bibliográficas: p. 77-79.

1. Reuso dinâmico de traços 2. Arquitetura de Processador I. França, Felipe Maia Galvão II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas da Computação. III. Título

Agradecimentos

A Deus, me dar a vida e fazer dela um eterno motivo de agradecimentos por todas as bênçãos que sempre recebi.

Aos meus pais, Rute e Timoteu, por serem meu alicerce em todos esses anos e terem a maior parcela de responsabilidade por eu ser a pessoa que sou e estar onde estou hoje.

Ao meu irmão Ozomatli, por toda a dedicação e torcida pra eu conquistasse mais essa etapa.

Ao meu amado Anderson, por todo o apoio, credibilidade, paciência, estímulo, compreensão, dedicação, amor, parceria e cumplicidade, nos vários momentos em que precisamos abdicar de muitas coisas para que este trabalho pudesse ser concluído.

Ao Programa de Engenharia de Sistemas e Computação da Universidade Federal do Rio de Janeiro, por ter aceitado o meu ingresso neste e permitido a conclusão de mais essa fase importantíssima de minha formação acadêmica.

Aos meus verdadeiros amigos, que me viram aos poucos me ausentar das ocasiões em que sempre estive presente, compreendendo que isto fazia parte de uma fase necessária e muito desejada da minha vida.

Ao Julio Cesar da Costa, que durante toda a minha permanência no programa teve a sensibilidade de compreender quão difícil é a tarefa de se obter um título de Mestre, cumprindo uma jornada de trabalho em paralelo e, em função disso, fazendo-me concessões que foram fundamentais para a conclusão deste trabalho de dissertação.

Ao colega Andrey Coppieters, pela boa vontade em dividir seus conhecimentos nos meus primeiros passos decisivos para o desenvolvimento deste trabalho.

Aos colegas da COPPE/Sistemas, mais especificamente do LAM, Bruno França, Leandro Marzullo e Lawrence Bandeira, pela atenção dispensada.

Ao meu orientador de fato, Felipe Maia Galvão França, por ter aceitado a parceria neste trabalho de dissertação e ter me tranqüilizado e garantido que tudo daria certo, em todos os momentos em que eu demonstrei insegurança, acreditando nas possibilidades que este trabalho traria.

Em especial, agradeço ao Professor Maurício Lima Pilla, meu "orientador de fé", por todas as horas de dedicação, auxílio, infindáveis explanações. Por dividir comigo seu conhecimento, que certamente me deu segurança e condições de continuar até o fim.

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

SENSIBILIDADE DO MECANISMO DE REUSO DE TRAÇOS AOS SUBCONJUNTOS DE INSTRUÇÕES

Sheila de Oliveira

Março/2009

Orientador: Felipe Maia Galvão França

Programa: Engenharia de Sistemas da Computação

Este trabalho apresenta um estudo sobre a sensibilidade do mecanismo do reuso de traços aos subconjuntos característicos de instruções. Fragmentar o domínio de instruções reusáveis em subconjuntos torna possível perceber a parcela de contribuição que cada um destes possui dentro do contexto do reuso de traços. A partir da criação de um Índice de Eficiência (IE), que é resultante da aceleração obtida em cada subconjunto sobre o percentual de instruções executadas neste subconjunto, foi possível avaliar de forma mais precisa quão importante cada subconjunto é dentro do mecanismo de reuso. Subconjuntos de instruções lógicas e aritméticas, instruções de desvio e instruções de acesso à memória foram criados e simulações foram realizadas com estes. Concluiu-se que instruções lógicas e aritméticas têm grande relevância dentro do mecanismo de reuso, no momento em que atingem 92,8% da aceleração do reuso total, com apenas 42,8% do percentual de reuso. Instruções de desvios apresentam o melhor IE, uma vez que atingem 90,3% da aceleração com um reuso de 14,5%. Instruções de Acesso à Memória não participam do reuso de traços, apenas de instruções isoladas, mas mesmo assim conseguem manter mais do que o dobro da eficiência do Reuso Total, usando o índice criado como referencial comparativo.

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

SENSITIVITY OF THE TRACE REUSE MECHANISM TO INSTRUCTION SUBSETS

Sheila de Oliveira

March/2009

Advisor: Felipe Maia Galvão França

Department: Engenharia de Sistemas da Computação

This work presents a study about the sensitivity of the trace reuse mechanism to instruction subsets. Fragmenting the reusable instructions domain in subsets makes it possible to understand the contribution of each subset within the trace reuse context. The creation of an Efficiency Index (EI), which is obtained from a division of each subset speedup by the percentage of instructions executed in that subset, allows for a more precise evaluation of the importance of each subset within the reuse mechanism. Subsets of logical and arithmetic instructions, branch instructions and memory access instructions were created and simulations were performed with them. The conclusions are: logical and arithmetic instructions are very important in reuse mechanism, since they alone produces 92.8% of total reuse speedup, with only 42.8% of reusable rate; branch instructions have the best EI, since they alone produces 90.3% of total reuse speedup, with 14.5% of reusable rate. Memory access instructions do not participate in trace reuse, only single instructions, but still produce more than twice the total reuse efficiency, using the index created as a comparative benchmark.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

BHB	Block History Buffer
DTM	Dynamic Trace Memoization
IE	Índice de Eficiência
LLT	Load Linking Table
LT	Load Table
RB	Reuse Buffer
RCB	Redundant Computation Buffer
RLQ	Recent Load Queue
RSQ	Recent Store Queue
RST	Reuse Through Speculation on Traces
RST	Register Source Table
RT	Recovery Table
RTM	Reuse Trace Memory

1. Introdução

Na área de Arquitetura de Computadores, todo o esforço está em melhorar o desempenho das computações de uma forma geral. Uma das vertentes está em alcançar avanços na tecnologia dos semicondutores, permitindo processadores com freqüências de *clock* cada vez maiores. Ainda no contexto de processadores, as evoluções no processamento com múltiplos núcleos também tende a resolver muitos desafios na área da computação. Outra frente de pesquisas nesta área são os aperfeiçoamentos de mecanismos, seja em nível de software ou hardware, que maximizem o número de instruções executadas por ciclo de *clock*.

O reuso de computações redundantes é uma das frentes de pesquisa que possui por objetivo obter ganhos de desempenho em processamento, partindo da idéia de que explorar um comportamento redundante pode reduzir o número de instruções que são executadas em um programa, no momento em que possuindo as mesmas entradas, as computações podem produzir os mesmos resultados e reusar uma computação é o mesmo que não executá-la.

A técnica do reuso dinâmico de instruções baseia-se em poder identificar dinamicamente que instruções possuem seus contextos de entrada reincidentes, possibilitando como conseqüência ter sua execução descartada, em função do reaproveitamento das saídas anteriormente produzidas. Para que tal mecanismo seja implementado, um conjunto de instruções pré-definidas como redundantes é estabelecido. Estas instruções devem ser então armazenadas em tabelas de reuso, de modo que, quando da repetição das entradas de uma instrução já gravada nesta estrutura de reuso, os valores serão consultados diretamente na tabela, evitando uma reexecução.

Reusar instruções gerou resultados satisfatórios, sendo o ponto de partida para a implementação do mesmo mecanismo, agora com glanularidade em nível de traços, que representam seqüências dinâmicas de instruções. O mecanismo DTM - *Dynamic Trace Memoization* (COSTA,2001), que permite o reuso de traços utilizando um domínio de instruções reusáveis pré-definido obteve ganhos de cerca de 8,5% sobre uma arquitetura sem mecanismo de reuso. Em seguida, o mecanismo RST - *Reuse Through Speculation on Traces* (PILLA,2004), que usou o DTM como arquitetura substrato, incorporando a

possibilidade de especulação de valores, atingiu uma melhora de em média 7% em desempenho com relação ao DTM.

Partindo do conhecimento das instruções pertencentes ao domínio de instruções reusáveis do DTM, este trabalho de dissertação objetivou fazer uma análise deste, quanto às suas contribuições ao desempenho obtido na técnica de reuso de traços, de forma segmentada. O domínio de reuso foi dividido em subconjuntos de reuso: subconjunto Add e Sub, representado pelo conjunto das instruções lógicas e aritméticas, o subconjuntos Desvios, que inclui todas as instruções de desvio condicional e incondicional; e o subconjunto Load e Store, que engloba as instruções de acesso à memória. Análises comparativas foram feitas em cada subconjunto de reuso, de forma a permitir uma análise da parcela de contribuição que cada subgrupo destes possui com relação a todo o domínio das instruções reusáveis.

Para fins comparativos, foram utilizadas as medidas de aceleração e a taxa de reuso de cada subconjunto criado. Um Índice de Eficiência (IE) foi criado, visando analisar a aceleração de forma conjunta com o percentual das instruções executadas em cada subconjunto. Este índice mede uma espécie de aceleração relativa, considerando o percentual de instruções executado.

O subconjunto de reuso Add e Sub apresentou um desempenho satisfatório, uma vez que atingindo 92,8% da aceleração do Reuso Total e usando cerca de 42,8% da taxa de reuso, obteve altos índices de eficiência, 2,27 vezes maior, se comparados com o Reuso Total.

Para o subconjunto de Desvios, os índices de eficiência foram os mais altos de todos os subconjuntos analisados. Tal situação é decorrente da taxa de reuso baixa, em torno de 4,7%, contra 31,8% na configuração de Reuso Total.

Analisando o subconjunto inverso dos desvios, ou seja, Tudo Menos Desvios, além do índice de eficiência 4,59 vezes melhor que o reuso total, taxa de reuso de 26,44% e aceleração satisfatórios, quando comparados ao Reuso Total, não reusar desvios também garante um menor espaço para alocação de memória para as tabelas de reuso. Eliminando os campos específicos de desvios nas tabelas de reuso, foi alcançada uma redução de 2,1% de espaço na Memo_Table_G e 10,7% em Memo_Table_T.

Dentro das avaliações de desempenho para o subconjunto Load e Store, detectou-se que não há reuso de traços dentro deste conjunto, devido a uma limitação do próprio mecanismo, que especifica as instruções de acesso à memória como delimitadoras na criação dos traços. Ainda assim, considerando somente instruções

isoladas, este subconjunto possui índice de eficiência 2,29 vezes melhor do que no Reuso Total.

Os resultados das experimentações permitiram elencar os subconjuntos que se destacam em aspectos diferentes. O ganho em aceleração de forma isolada confirma o Reuso Total como a melhor forma de configuração de reuso. Já considerando a redução do consumo de energia, os subconjuntos Load e Store para reuso de instruções isoladas e o subconjunto de Desvios para reuso de traços seriam as escolhas mais adequadas. Outro ponto importante que pode ser considerado ao desenvolver uma arquitetura que implemente o reuso de traços é o espaço de alocação de memória dedicado a este mecanismo de reuso. Caso a opção seja alocar o mínimo de espaço possível, eleger o subconjunto Load Store representaria uma economia de 45,6% em alocação de espaço em memória, se comparado à configuração de Reuso Total, e o subconjunto Tudo Menos Desvios, apresenta redução de 12,8% de espaço de memória, com a exclusão dos campos específicos do reuso de desvios em Memo_Table_G e Memo_Table_T. Finalmente, se um projetista considerar o Índice de Eficiência criado neste trabalho como ponto de decisão, o subconjunto mais eficiente seria o subconjunto de Desvios.

Este trabalho de dissertação divide-se em 6 capítulos, incluindo este de Introdução. No Capítulo 2, os trabalhos relacionados à área de reuso são apresentados. Os trabalhos de reuso em nível de traços, que serviram de base para este trabalho de dissertação são mais detalhadamente explanados no Capítulo 3. No Capítulo 4, toda a parte experimental, bem como os resultados encontrados nas simulações são expostos e analisados. O Capítulo 5 explicita as conclusões deste trabalho. Por fim, as Referências Bibliográficas estão relacionadas no Capítulo 6.

2. Trabalhos Relacionados

Neste capítulo serão apresentados os trabalhos relacionados a pesquisas na área de computação redundante e reuso, em suas diversas granularidades. O início do capítulo apresenta na Seção 2.1 o Reuso de Instruções Dinâmicas. Em seguida, na Seção 2.2 o Result Cache é apresentado. Na Seção 2.3, será introduzido o conceito da Remoção Dinâmica de Computações Redundantes. Posteriormente, na Seção 2.4, será descrita a técnica de Reuso de Blocos Básicos. Em seguida, o Reuso de Sub-Blocos será apresentado na Seção 2.5. Na Seção 2.6, será explanada a técnica de Reuso em Nível de Traços e finalmente, na Seção 2.7, a técnica do Reuso de Load encerrará o capítulo dos trabalhos relacionados.

Os trabalhos de DTM e RST serão discutidos mais detalhadamente no Capítulo 3, uma vez que compõem a arquitetura base para os experimentos deste trabalho de Dissertação.

2.1. Result Cache

Um estudo de computação redundante, baseado na natureza trivial das computações foi realizado. Por computação trivial entende-se que são todas as operações complexas como multiplicações e divisões, por exemplo, que podem ser trivializadas, como as operações e divisões por elementos neutros dessas operações. Trivializando uma computação qualquer, é possível obter aceleração na execução de um determinado programa (RICHARDSON,1992).

2.1.1. A natureza redundante da computação

Uma computação qualquer envolve tipicamente um conjunto de dados de entrada inicial, a transformação destes dados em um ou mais estados e dados finais de saída. Alguns desses dados de entrada são de natureza redundante e por isso tendem a ter seu fluxo passando por estados similares. Programas freqüentemente executam

múltiplas vezes com a mesma entrada ou entradas similares e o conhecimento desta natureza redundante pode acelerar a tarefa da computação de muitas formas. O acesso à memória cache, por exemplo, funciona tão bem porque as mesmas áreas de memória são acessadas muitas vezes em um período curto de tempo. Outro exemplo a ser mencionado é a compilação incremental, que tira vantagem do fato de que programas em desenvolvimento raramente variam muito de uma execução para a outra (RICHARDSON,1992).

- **Memoização**

A técnica de memoização ou tabulação tira vantagem da natureza redundante da computação. Ela permite a um programa ser executado mais rápido através da troca do tempo de execução pelo aumento da capacidade de memória. Uma vez calculado, o resultado de uma função é armazenado em uma tabela chamada Memoization Cache. A cache existe tradicionalmente como uma estrutura de dados. A busca da cache substitui então chamadas à função posteriores. A tabulação pode ser estendida para ser aplicada não só a funções, mas a declarações, grupos de declarações, ou até mesmo qualquer região de um programa que tem efeitos colaterais limitados e alto grau de recorrência (RICHARDSON,1992).

Aplicações eficientes desta técnica requerem que seja encontrada uma declaração muito usada no programa e a maior região de código que possui efeitos colaterais limitados e usar o rastreamento do valor para verificar se uma recorrência significante pode ocorrer (RICHARDSON,1992).

- **Result Caching**

Para esta técnica, um hardware especial para cache poderia executar a tabulação sem a necessidade de intervenção do compilador e nem do programador. O acesso a este *Result Cache* poderia ser iniciado ao mesmo tempo, que uma divisão de ponto flutuante, por exemplo. Se o acesso à cache resultar em um acerto, a resposta aparece rapidamente e a operação de ponto flutuante pode ser interrompida. Em um erro, a unidade de

divisão pode gravar o resultado no cache ao mesmo tempo em que envia o resultado para a próxima fase do pipeline (RICHARDSON,1992).

As duas técnicas apresentam melhores resultados em programas que fazem uso intensivo de operações com ponto flutuante, provavelmente porque a maioria destas operações possui alta latência. Em razão de tal fato, qualquer instrução de longa latência torna-se candidata a ganhar aceleração, utilizando estas técnicas (RICHARDSON,1992).

2.2. Reuso de Instruções Dinâmicas

As memórias caches reduzem o número de acessos à memória principal feitos dinamicamente, se uma determinada localização de memória for acessada repetidamente. A partir desta idéia, foi desenvolvido um mecanismo de reuso de instruções dinâmicas. Da mesma maneira que a cache reduz os acessos repetidos, devido à localidade de memória, o número de instruções executadas dinamicamente pode ser reduzido, se estas instruções produzirem o mesmo valor repetidamente (SODANI;SOHI,1997).

Muitas instruções e grupos de instruções que têm a mesma entrada, quando são executados dinamicamente, podem reduzir o seu número de execuções, se o resultado anterior de sua instrução for guardado em alguma estrutura de buffer, de modo que outras instâncias dinâmicas futuras desta mesma instrução estática possam usar o resultado, estabelecendo que os operandos de entrada em ambos os casos são os mesmos. Tal operação é conhecida por reuso dinâmico de instruções (SODANI;SOHI,1997).

O reuso dinâmico de instruções pode se beneficiar do desempenho de duas maneiras. Primeiro, por não passar por todas as fases de execução dinamicamente, a utilização dos recursos de máquina pode ser reduzida, diminuindo os conflitos de recursos. A segunda forma é conhecer a saída de uma instrução previamente, possibilitando a antecipação da execução de instruções dependentes, ou seja, os resultados das cadeias de instruções dependentes podem ser gerados em um único ciclo, reduzindo o tamanho dos caminhos críticos de execução (SODANI;SOHI,1997).

2.2.1. Esquemas para o Reuso de Instruções

Sodani e Sohi implementaram três esquemas em hardware para o reuso de instruções dinâmicas. Em cada esquema, o resultado de execuções prévias é armazenado em uma estrutura chamada de *Reuse Buffer* (RB). O projeto do Reuse Buffer é apresentado na Figura 2.1. Quando uma instrução é executada, o RB é consultado para verificar se existe algum resultado reusável para esta instrução.

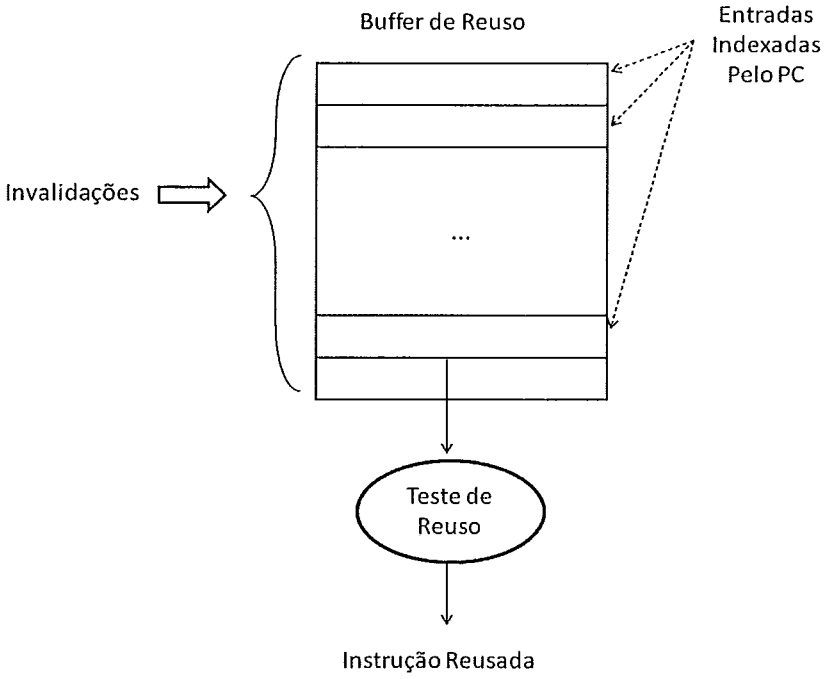


Figura 2. 1 – Reuso Buffer Genérico

A informação no RB é acessada através do PC da instrução e esta informação pode ser organizada de maneira associativa. Quanto maior a associatividade, maior o número de instâncias dinâmicas que uma instrução pode manter na RB por um determinado tempo. Para saber se uma entrada RB tem informação reusável, um teste de reuso verifica se a informação acessada do RB possui um resultado reusável.

O primeiro esquema de teste (Sv) monitora os valores dos operandos de cada instrução, o segundo (Sn) monitora o nome dos operandos (identificador dos

registradores) de cada instrução e o terceiro (Sn+d) monitora as relações de dependência entre as instruções (SODANI;SOHI,1997).

- **O Esquema Sv**

No esquema Sv, os valores dos operandos de uma instrução são armazenados juntamente com o seu resultado. No momento da decodificação de uma instrução, os valores de seus operandos são comparados com os operandos que estão gravados no RB. Se eles forem os mesmos, os resultados guardados no RB são reusados. Para instruções load, o cálculo de endereço pode ser reusado se os operandos para o cálculo do endereço não mudarem. Entretanto, a saída real deste load só pode ser reusada se a posição de memória endereçada não tiver sido gravada por uma instrução *store*. O tratamento para instruções store também é diferenciado. Enquanto o reuso do cálculo de endereço do store não apresenta problemas, não há tentativa de reusar a gravação real da memória, uma vez que a gravação em memória pode ter efeitos colaterais (SODANI;SOHI,1997).

Tag	Operand1 value	Operand2 value	Address	Result	Mem valid

Figura 2. 2 - Estrutura da entrada RB para o Esquema Sv

Para cada entrada Sv, ilustrada na Figura 2.2 (SODANI;SOHI,1997), o campo ***tag*** grava parte do PC. Os campos ***result***, ***operand value 1*** e ***operand value 2*** gravam o resultado e os valores dos operandos da instrução, respectivamente. Esses campos são usados para identificar se a instrução pode ser reusada. O bit ***memvalid*** e o campo ***address*** são usados para determinar se o acesso real da memória para uma instrução load pode ser reusado; o bit ***memvalid*** indica se o valor carregado da memória (presente no campo result) é válido e o campo ***address*** grava o endereço na memória.

Para a verificação da possibilidade de reuso do esquema Sv, os operandos da instrução que está sendo executada são comparados com os valores dos campos ***operand value*** da entrada RB. Um acerto indica que o endereço é válido para instruções

que fazem acessos à memória, como load e store ou que o resultado é válido, para todas as instruções que não fazem acessos à memória. Quando a instrução corrente é um load, além de testar a validade dos bits de endereço, é necessário testar o bit ***memvalid***, para verificar se a saída do load pode ser reusada. Se os valores dos operandos não são conhecidos no momento do teste de reuso, a instrução não é reusada.

As invalidações para instruções que não acessam a memória não são necessárias para manter a integridade do teste de reuso, uma vez que os operandos determinam unicamente o resultado. Para instruções load, o valor do campo ***result*** é invalidado, se houver um store para o mesmo endereço. Conseqüentemente em um store, o campo ***address*** de cada entrada RB é procurado, objetivando encontrar um endereço correspondente e o bit ***memvalid*** é reiniciado para as entradas localizadas na busca.

Os campos ***address*** e ***memvalid***, além da busca associativa por invalidações, são requeridos apenas para manter a integridade dos valores de load. O RB pode ser dividido em dois buffers: um para gravar os valores armazenados de load, e outro, que é o RB principal, para gravar tudo, exceto os valores de load. Essa organização do RB tem duas vantagens. A primeira é que os campos ***memvalid*** e ***address*** não precisam ser mantidos para entradas de instruções que não acessam a memória, o que reduz o espaço de armazenamento para este esquema de reuso e a segunda vantagem é que o RB principal não precisa ter uma lógica de invalidação, uma vez que esta lógica só estaria presente no buffer para valores de load, que provavelmente serão muito menores se comparados ao valor principal (SODANI;SOHI,1997).

- **O Esquema Sn**

Neste esquema, o teste de reuso é mais simples, assim como cada entrada do RB é reduzida, uma vez que ao invés de gravar os valores dos operandos, são gravados os identificadores de registrador dos operandos no RB. Quando uma instrução faz uma gravação no registrador, todas as instruções que possuem os mesmos identificadores de registrador de origem no RB são invalidadas.

Tag	Operand1 reg name	Operand2 reg name	Address	Result	Result valid	Mem valid

Figura 2. 3 - Estrutura da entrada RB para o Esquema Sn

A principal diferença entre o esquema Sv e o esquema Sn é que os campos ***operand1*** e ***operand2*** contêm os nomes dos registradores dos operandos ao invés do valor dos operandos, conforme apresentados na Figura 2.3 (SODANI;SOHI,1997). Além disso, um bit ***resultvalid*** indica se o resultado é válido.

No esquema Sv o teste de reuso é realizado apenas verificando o estado dos bits ***memvalid*** e ***resultvalid***. O cálculo de endereço para instruções load/store e os resultados para todos os tipos de instruções podem ser reusados, caso o bit ***resultvalid*** seja ativado. Já o resultado de uma instrução load pode ser reusado se ***resultvalid*** e ***memvalid*** estão ativados.

Assim como no esquema anterior, instruções store invalidam os loads que acessam o mesmo endereço gravado anteriormente por esta instrução. Quando o registrador é gravado, uma busca por entradas cujo campo ***operand*** corresponde ao nome do registrador e as entradas onde ocorre o acerto na busca são marcadas como inválidas.

- **O Esquema Sn + d**

O esquema Sn+d complementa o esquema Sn na tentativa de estabelecer cadeias de instruções dependentes e monitorar o status de reuso de tais cadeias de instruções.

As cadeias de dependências são criadas através de entradas inseridas no RB. A fim de facilitar esse processo, é utilizada uma tabela de mapeamento denominada Register Source Table (RST). A RST tem uma entrada para cada registrador; ela rastreia a entrada RB que tem o último resultado para aquele registrador. Quando uma entrada é reservada no RB para uma instrução, a entrada RST para o seu registrador de destino é atualizada para apontar para esta entrada reservada. Se a instrução que é o último produtor de um registrador não está no RB, então a entrada RST para aquele registrador é ajustada como inválida. A RST funciona de forma similar à renomeação de registradores. Em essência, a RST é usada para fazer a ligação da instrução consumidora à última instrução produtora, através da ligação com o registrador (entrada RB) do produtor (SODANI;SOHI,1997).

As entradas do RB para o esquema Sn+d são similares ao esquema Sn, exceto pela adição do campo ***src-index***, como mostra a Figura 2.4 (SODANI;SOHI,1997). As cadeias de dependência são criadas armazenando o índice RB nas instruções de origem neste campo. Um valor inválido é inserido neste campo, caso a origem não exista no RB.

Tag	Operand		Operand2		Address	Result	Result valid	Mem valid
	Src-index	Reg name	Src-index	Reg name				

Figura 2. 4 – Estrutura da entrada RB para o Esquema Sn+d

Para a realização do teste de reuso no esquema Sn+d, o status do reuso de instruções independentes é obtido da mesma forma que no esquema Sn. Uma instrução dependente é reusada se suas instruções de origem no RB, indicadas pelo campo ***src–index*** dos seus operandos, são realmente os últimos produtores para os seus operandos.

Para as invalidações neste esquema, da mesma maneira que nos esquemas Sv e Sn, os stores invalidam os loads para o mesmo endereço. Como no esquema Sn, instruções independentes são invalidadas quando seus registradores são sobrescritos. Instruções dependentes não precisam ser invalidadas na sobrescrita dos operandos, uma vez que seu status de reuso pode ser estabelecido através de sua informação de dependência. Ao invés disso, eles são invalidados quando suas instruções de origem são descartadas do RB, ou seja, quando a informação de dependência é perdida. Para executar essa operação, é necessário procurar por entradas cujo campo ***src–index*** corresponde ao índice no RB da instrução de origem que está sendo descartada. Todas as entradas que correspondem ao resultado desta busca são invalidadas (SODANI;SOHI,1997).

2.2.2. Micro arquitetura com um Reuse Buffer

A unidade de Busca de Instruções procura e armazena as instruções na Fila de Instruções. A decodificação de instruções e renomeação de registradores são feitas na unidade de Decodificação e Renomeação. No estágio de decodificação e renomeação o RB é acessado para verificar se o resultado reusável para instrução pode ser encontrado.

Em caso positivo, esta instrução não precisará ser executada; ela não entra na Janela de Instruções e prossegue diretamente para o Buffer de Reordenação. Instruções Load não são incluídas na Janela de Instruções, a não ser que micro operações, cálculos de endereço e operações reais de memória possam ser reusadas. Se um resultado reusável não é encontrado no RB, uma entrada é então reservada no RB, onde o resultado da instrução será alocado depois de ser executado. Uma vez na Janela de Instruções, as instruções prosseguem como se estivessem em qualquer processador escalar genérico. Depois que a instrução é executada, seus resultados são armazenados na entrada RB reservada. No esquema Sv, os valores dos operandos também são gravados na entrada RB nesse momento. Quando uma instrução é finalizada, dependendo do seu esquema de reuso, ela invalida os resultados apropriados no RB. A Figura 2.5 mostra uma microarquitetura genérica com um RB.

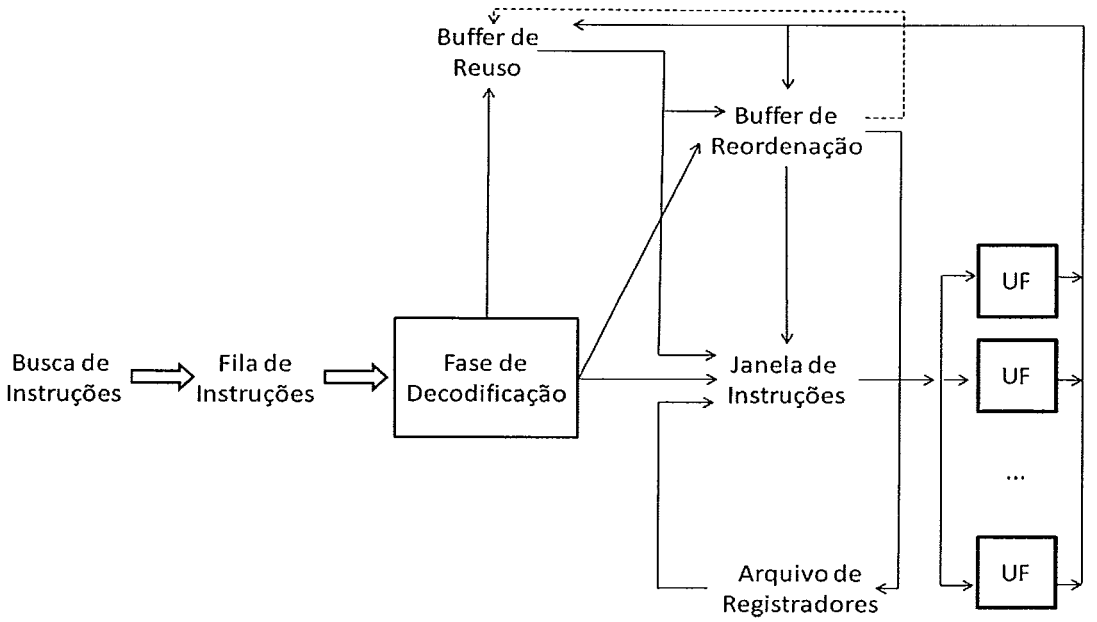


Figura 2. 5 - Microarquitetura genérica com um Reuse Buffer

Uma vez que o RB contenha o estado que vai determinar a saída de futuras instruções, ele precisa ser mantido com precisão. Uma maneira simples de fazer isso é atualizar o RB somente quando uma instrução é entregue. Essa abordagem previne que instruções executadas especulativamente entrem no RB. Para o esquema Sv, a inserção de instruções no RB especulativamente não requer ações especiais – o teste de reuso

garante que o resultado é obtido. No esquema Sn+d, a RST controla a reusabilidade das instruções (SODANI;SOHI,1997).

2.3. Remoção Dinâmica de Computações Redundantes

Molina, Gonzalez e Tubella (1999) analisaram o contexto de reuso dinâmico de instruções. Segundo os autores, uma instrução dinâmica pode reusar um resultado de uma instância prévia da mesma instrução estática ou uma instância de outra instrução estática. No caso de um reuso para uma mesma instrução estática, o que ocorre é a remoção daquilo que é conhecido como "quase-invariante", ou seja, uma computação que é repetida muitas vezes e produz quase sempre o mesmo resultado. Para o reuso de instâncias de outras instruções estáticas, a situação corresponde à eliminação de uma "subexpressão quase-comum", que é uma computação que freqüentemente produz o mesmo resultado de outra parte do código. No exemplo ilustrado da Figura 2.6(a) (MOLINA; GONZALEZ; TUBELLA,1999), se os vetores b e c tiverem muitos elementos repetidos, a computação b[i] + c[i] vai se tornar quase-invariante em tempo de execução. Similarmente, se t e u têm o mesmo valor, em 2.6(b) (MOLINA; GONZALEZ; TUBELLA,1999), então as expressões s/t e s/u serão subexpressões quase comuns.

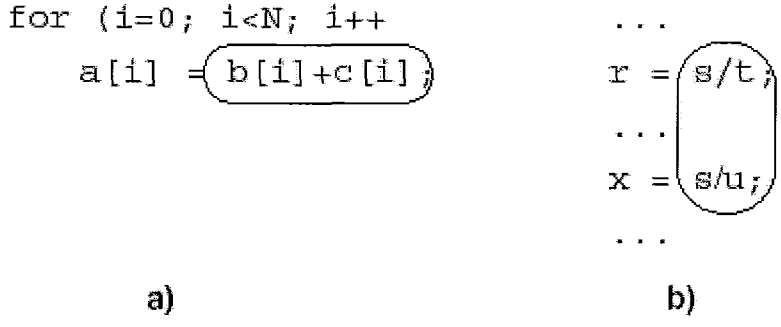


Figura 2. 6 - Exemplos de Quase-Invariantes (a) e Subexpressões Quase Comuns (b)

Este trabalho propôs um mecanismo que ao mesmo tempo remove quase-invariantes em tempo de execução e elimina as subexpressões quase-comuns.

O reuso em nível de instrução tem duas vantagens: reduz a latência de algumas instruções e reduz a contenção dos recursos do processador, uma vez que instruções reusadas não passam pelas fases de emissão e execução do pipeline. A diferença entre esta técnica e otimizações feitas pelo compilador é que o compilador só pode remover algumas das computações redundantes, devido ao seu conhecimento limitado dos dados e usualmente não identifica computações quase-redundantes (MOLINA; GONZALEZ; TUBELLA,1999).

2.3.1. O Redundant Computation Buffer

A Figura 2.7 ilustra o mecanismo que explora o reuso em nível de instrução é baseado em um buffer chamado Redundant Computation Buffer (RCB), que armazena a informação sobre as computações que são suscetíveis a serem redundantes.

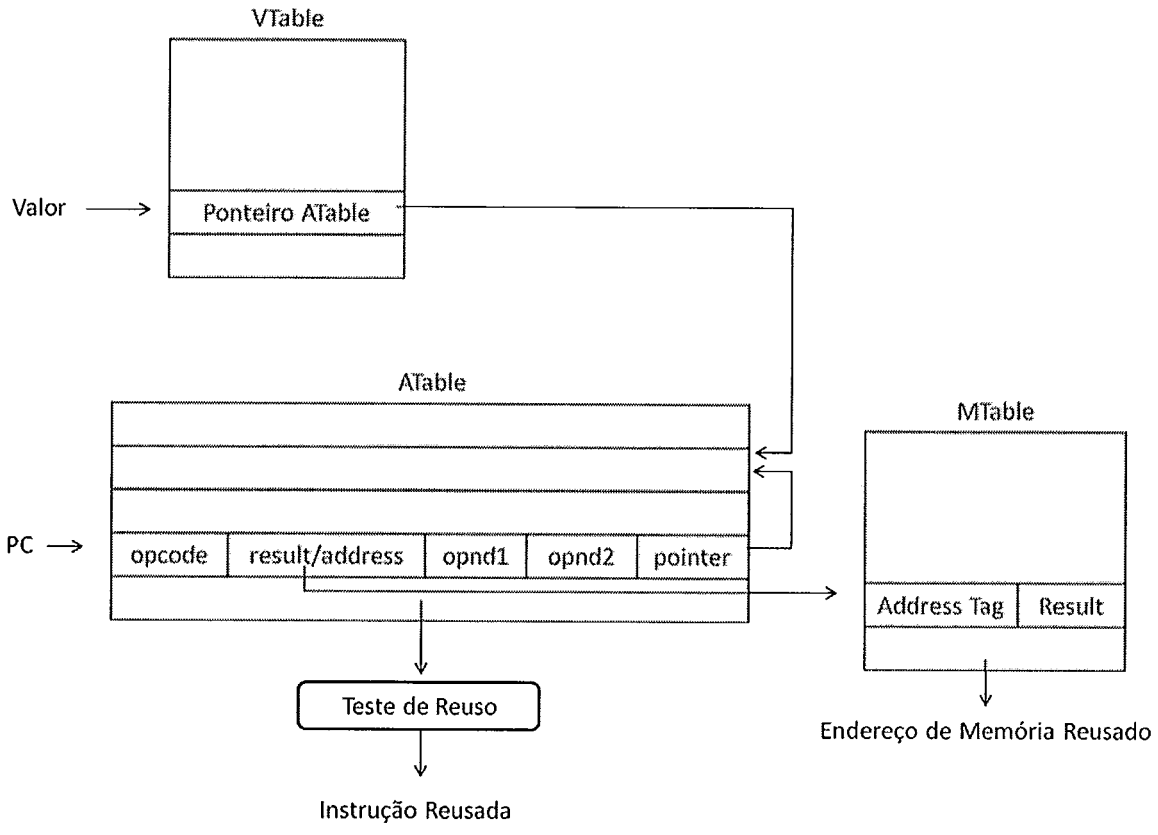


Figura 2. 7 - Estrutura do Redundant Computation Buffer

Uma importante diferença entre o Result Cache (RICHARDSON,1992) e o Reuse Buffer (SODANI;SOHI,1997) é que o primeiro necessita dos operandos de entrada para indexar o buffer, enquanto o segundo é indexado pelo endereço da instrução e seus operandos só são requeridos quando a entrada é lida, a fim de serem comparados com os operandos da entrada do buffer. Em outras palavras, o Reuse Buffer pode sobrepor a pesquisa ao buffer com a busca de instruções, renomeação de registradores e operações de leitura do registrador, enquanto o Result Cache não. Portanto, se assumirmos que a pesquisa na tabela consome um ciclo, o mecanismo de Reuse Buffer pode produzir um resultado de uma instrução reusável antes que o Result Cache. Por outro lado, a principal desvantagem do Reuse Buffer é que ele só pode reusar dinamicamente instâncias da mesma instrução estática, ou seja, o Reuse Buffer procura explorar as quase-invariantes, mas não tira vantagem das subexpressões quase-comuns. Mais ainda, o Reuse Buffer pode reusar múltiplas instruções buscadas simultaneamente e dependentes entre si. Uma vez que o Reuse Buffer é indexado pelo endereço da instrução, múltiplas entradas correspondentes podem ser lidas simultaneamente e o resultado reusado de uma instrução pode ser usado para checar se as instruções dependentes seguintes podem ser reusadas, de forma similar à renomeação de registradores de múltiplas instruções dependentes, que pode acontecer no mesmo ciclo. Essa característica é chamada de encadeamento de reuso, que não pode ser explorada pelo Result Cache, uma vez que a busca no buffer é seqüencial e não pode ser executada em um único ciclo (MOLINA; GONZALEZ; TUBELLA,1999).

O RCB procura eliminar a computação redundante resultante das quase-invariantes e das subexpressões quase-comuns. Além disso, o RCB é indexado pelo endereço da instrução, tendo, portanto a mesma latência do Reuse Buffer e pode explorar o encadeamento de reuso (MOLINA; GONZALEZ; TUBELLA,1999).

Quando duas instruções estáticas produzem os mesmos valores, a que os produz posteriormente pode reusar a computação executada pela anterior. Neste caso, a primeira instrução é o consumidor e a outra é o produtor (MOLINA; GONZALEZ; TUBELLA,1999).

O RCB consiste de três tabelas: a primeira armazena os resultados de reuso de instruções aritméticas e endereços de memória (Atable); a segunda tenta reusar através de teste de reuso os valores de Load (Mtable) e a terceira tabela é usada para identificar as subexpressões quase-comuns (Vtable). A Atable é indexada pelo endereço da

instrução e cada entrada contém os seguintes campos: ***opcode***, que especifica a operação que será executada; ***result/address***, que corresponde tanto ao resultado da última instrução aritmética quanto ao endereço da última operação de memória que foi mapeada para aquela entrada; ***opnd1 e opnd2***, que corresponde aos valores dos dois operandos da instrução e o ***pointer***, que é um ponteiro para a entrada que armazena os resultados da instrução, que produz valores que podem ser reusados pela instrução mapeada pela entrada, ou seja, um ponteiro para o produtor (MOLINA; GONZALEZ; TUBELLA,1999).

As entradas não incluem uma tag. Uma instrução pode reusar o resultado de uma entrada, desde que os operandos de entrada e o opcode correspondam àquela entrada, não importando a que instrução estática aquela entrada corresponda. Isto não só reduz o tamanho da tabela, como também permite tirar vantagem de interferências entre instruções que tenham o mesmo opcode (MOLINA; GONZALEZ; TUBELLA,1999).

A Mtable é indexada pelo endereço efetivo de operações de memória, que são obtidas pela Atable e cada entrada contém o último valor lido ou escrito para aquela posição, além da tag que identifica o endereço efetivo. A tabela age de forma similar a uma memória cache e não é parte essencial do mecanismo. O benefício principal de tal tabela, em comparação ao acesso à hierarquia de memória é o tempo de acesso menor, uma vez que a tabela possui menor capacidade e a redução da pressão das portas cache.

A Vtable armazena informação sobre os últimos resultados das instruções. É indexado pelo valor do resultado e cada entrada pode conter o opcode e o endereço da instrução que produziu aquele valor.

O RCB funciona da seguinte forma. A Atable é acessada duas vezes, enquanto as instruções são buscadas e decodificadas. A primeira busca usa o endereço da instrução e obtém o resultado potencial, se a instrução produz o mesmo resultado que a última execução da mesma instrução estática, ou outra instrução estática mapeada para a mesma entrada. Neste momento, o ponteiro para a instrução produtora também é obtido, no caso do reuso ser de outra instrução estática. Os últimos operandos e o resultado do produtor são resgatados quando a tabela é acessada pela segunda vez. Uma vez que a instrução decodificada e seus operandos de entrada tenham sido lidos, os operandos de entrada reais são comparados com os últimos operandos da mesma instrução estática e o opcode da instrução produtora. Se uma dessas comparações corresponderem, a instrução desvia os estágios de emissão e execução e prossegue diretamente para o estágio de write-back.

As instruções Load podem reusar um endereço anteriormente calculado, usando Atable. Além disso, elas podem reusar seus valores de load por meio da Mtable. Esta última é acessada pelas instruções load que conseguiram reusar seus endereços e são indexadas por estes. Instruções store também podem usar seus endereços efetivos por meio da Atable. Os loads atualizam a Mtable quando executam e gravam os valores no término da execução.

A Vtable, que é indexada pelo valor de resultado, é atualizada por toda instrução, no momento do término da execução. Antes de gravar nesta tabela, toda instrução primeiro lê a entrada correspondente e checa se esta entrada foi previamente atualizada por outra instrução com o mesmo opcode. Neste caso, um ponteiro do consumidor (instrução corrente) para o produtor (instrução encontrada em Vtable) é ajustado em Atable (MOLINA; GONZALEZ; TUBELLA,1999).

2.3.2. Result Cache e Reuse Buffer Modificados

Os esquemas anteriores permitem que o processador reuse o resultado da última execução das instruções. Entretanto, em alguns casos, instruções podem reusar resultados de execuções prévias tanto da mesma instrução, como de outra instrução estática. Esta situação pode ser explorada com o uso de um histórico que armazena os N últimos resultados de cada instrução. O N corresponde à profundidade do histórico (MOLINA; GONZALEZ; TUBELLA,1999).

O Result Cache consiste de um buffer que é indexado pelo hashing dos valores dos operandos de entrada e o opcode. Cada entrada contém os operandos de entrada e o seu resultado. O Result Cache também possui uma tabela que grava os valores de memória, que têm a mesma estrutura do Mtable no esquema RCB.

O Reuse Buffer consiste de duas tabelas. Uma tabela é usada para instruções aritméticas e valores de memória e a outra tabela é utilizada para a busca de reuso de valores de memória. Em sua versão original, as duas tabelas são mescladas em uma única. Pelo fato de implementar tabelas divididas representar uma solução mais custo efetiva, a versão modificada possui esta tabela separada para instruções de memória, que é indexada pelo endereço da instrução. A implementação requer que as instruções store procurem associativamente a tabela em busca do endereço de memória (MOLINA; GONZALEZ; TUBELLA,1999).

O Reuse Buffer pode ser melhorado com algumas características do RCB. A versão melhorada inclui as seguintes otimizações:

- Tem buffers separados, como no RCB. Um é destinado a resultados aritméticos e endereços de memória e outro para valores de memória
- A tag com o endereço da instrução não é incluída. Isto permite reduzir a quantidade de armazenamento requerido.
- Passa a habilitar o reuso para execuções mais anteriores de instruções estáticas, com uso da profundidade de histórico.

O Result Cache pode explorar o reuso tanto de quase-invariantes quanto de subexpressões quase-comuns. Além disso, o Result Cache pode explorar o reuso de resultados de execuções mais prévias. Por outro lado, o Reuse Buffer só pode explorar quase-invariantes. Além disso, como é indexada pelo endereço da instrução, a latência do reuso do Reuse Buffer é menor do que no Result Cache (MOLINA; GONZALEZ; TUBELLA,1999).

2.4. Reuso de Blocos Básicos

Uma técnica de implementação de reuso foi desenvolvida, com granularidade de blocos básicos, baseados na idéia de que entradas e saídas de uma cadeia de instruções são altamente correlacionadas. Um bloco básico pode ser visto como uma sequência de instruções que só possui uma única entrada e uma única saída. Usar o bloco básico como uma unidade de predição e reuso pode economizar hardware, se comparado ao reuso em nível de instrução, além de reduzir o tempo de execução (HUANG;LILJA,1999).

Huang e Lilja (1999) investigaram a localidade dos valores da entrada e saída dos blocos básicos para determinar sua previsibilidade e potencial de reuso. Os limites do bloco básico são determinados dinamicamente em tempo de execução e as entradas de cada bloco básico são gravadas em um novo mecanismo de hardware chamado Block History Buffer, ilustrado na Figura 2.8 (HUANG;LILJA,1999). O processador usa esses valores gravados para determinar os valores de saída que um bloco básico vai produzir na próxima vez que for executado. Se as entradas atuais do bloco forem as mesmas da

última vez que o bloco foi executado, todas as instruções no bloco podem deixar de ser executadas. Esta técnica é chamada de reuso de bloco.

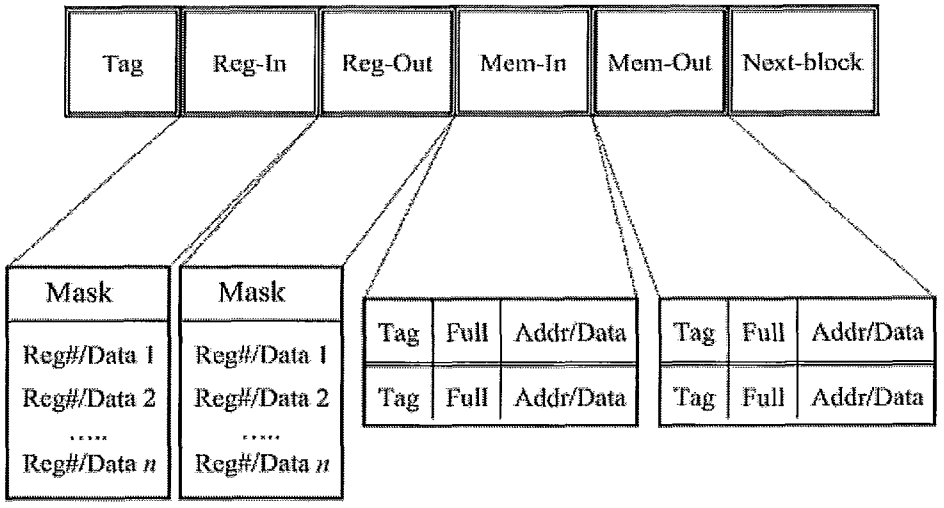


Figura 2. 8 - Estrutura de uma entrada do Block History Buffer

2.4.1. Implementação

Os valores de entrada e de saídas devem ser armazenados para cada bloco básico no BHB, juntamente com o endereço inicial do próximo bloco básico. Quando o ponto de entrada para um bloco básico é encontrado na execução de um programa, o BHB é checado para verificar se a saída deste bloco é determinável, ou seja, se todos os valores de entrada para o bloco, incluindo qualquer entrada de memória armazenada em cache, corresponderem aos valores gravados no BHB, o processador vai para o bloco subseqüente e deixa de executar todo o trabalho do bloco atual. Se a saída deste bloco não é determinável, o processador envia instruções para a unidade funcional como é o procedimento normal. Quando qualquer instrução que está no bloco básico termina sua execução, o BHB é atualizado. A Figura 2.9 apresenta o modelo do processador.

Os blocos básicos são construídos dinamicamente usando os seguintes passos:

a) Qualquer instrução depois de um desvio é identificada como um ponto de entrada de um novo bloco. A primeira instrução de um programa é o ponto de entrada de um bloco automaticamente. Chamadas de subrotinas

e retornos são tratados exatamente como qualquer outro tipo de instrução de desvio.

b) A execução de uma instrução de desvio marca o final de um bloco básico.

c) Um desvio no meio do bloco básico divide o bloco atual em dois blocos separados.

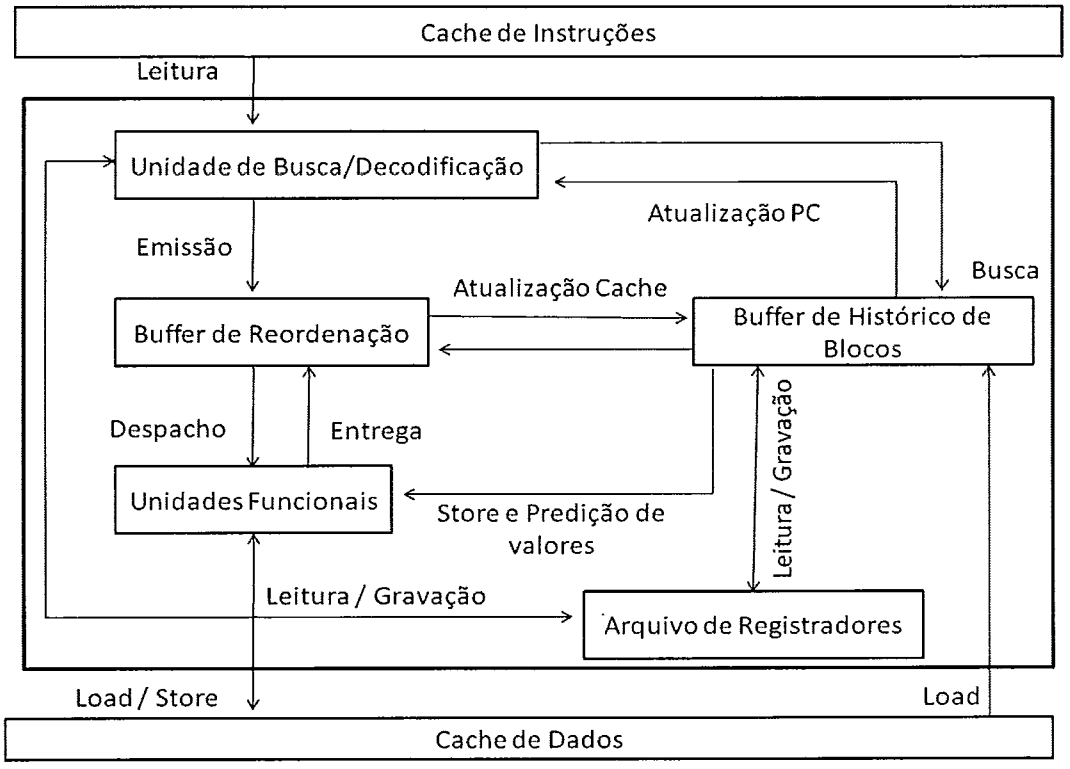


Figura 2. 9 - Modelo do processador que avalia o potencial de desempenho do reuso de blocos

Quando uma instrução é buscada, o BHB é consultado. Se a instrução corresponde a uma entrada para o bloco no BHB, os valores atuais de entrada para o bloco básico são comparados com os valores no buffer, no momento em que a instrução alcança o estágio de emissão, ou seja, quando todos os operandos estão prontos. Quando qualquer entrada de memória no bloco básico é válida, os dados da cache devem ser acessados. Se o acesso produz um acerto, o valor da cache é comparado com os que estão gravados no BHB. Se o acesso produz um erro, verifica-se que o conteúdo da memória é diferente e a localidade de valor é perdida. Durante o processo de comparação, o processador continua a sua execução normal. Então, o tempo de execução que foi economizado pelo reuso do bloco precisa ser compensado pelo tempo

requerido pela comparação dos operandos, para produzir aceleração (HUANG;LILJA,1999).

O hardware coleta os valores de entrada e saída dos blocos básicos dinamicamente. Quando uma instrução é executada, os bits da máscara de entrada para todos os registradores lógicos de entrada são ajustados e os bits da máscara de saída apropriados são ajustados para os registradores de saída dos blocos. Quando o BHB determina que todas as instruções no bloco são redundantes e podem deixar de ser executadas, as ações são executadas de acordo com o tipo de processamento de exceção desejado. Para exceções bem definidas, por exemplo, as instruções são emitidas como num processamento normal. Elas são marcadas como completas quando elas reservam entradas no Reorder Buffer, que evita o consumo de recurso de qualquer unidade funcional (HUANG;LILJA,1999).

Registradores são freqüentemente usados para gravar os resultados intermediários para todos os tipos de operações nos programas. Contudo, esses resultados intermediários são raramente usados fora dos blocos básicos que os produzem. Resultados que são produzidos dentro de um bloco básico, mas nunca usados nos blocos básicos seguintes são saídas mortas e devem ser excluídas das saídas dos blocos. Apesar de ser possível utilizar o hardware para distinguir saídas mortas, dentro do escopo de alguns blocos básicos consecutivos na janela de execução de instruções, pode ser não realista para o hardware identificar todas as saídas que nunca serão usadas em caminhos de execução subseqüentes. O compilador, entretanto, pode alcançar esta tarefa usando a análise do fluxo de dados (HUANG;LILJA,1999).

Para arquiteturas load-store, os endereços de memória mudam apenas quando os registradores de entrada usados para calcular os endereços também mudam. O BHB checa o conteúdo da cache de dados, assim como os endereços que estão sendo referenciados. Conseqüentemente, mesmo se o programa usa níveis múltiplos de ponteiros, o BHB ainda detecta a repetição de entradas de blocos corretamente (HUANG;LILJA,1999).

2.5. Reuso de Sub-Blocos

Após a implementação do reuso em nível de blocos básicos, que consiste em um conjunto de instruções que contém um ponto de entrada e um ponto de saída, Huang e

Lilja verificaram que o número de instruções, assim como o número de entradas e saídas de um bloco básico, podem ser ilimitados, uma vez que o final de um bloco básico geralmente é delimitado pela presença de uma instrução de desvio. Por este motivo, é impossível para uma estrutura finita de hardware capturar todos os blocos básicos em programas diferentes. Os sub-blocos, que são criados pela divisão dos blocos básicos maiores, têm tamanhos gerenciáveis e controláveis, bem como um número limitado de entradas e saídas. Similar aos blocos básicos, sub-blocos são seqüências estáticas de instruções. Eles também são determinados, analisados e otimizados em tempo de compilação. Por isso, sub-blocos podem ser uma melhor alternativa para servir como unidade de reuso que o bloco básico (HUANG;LILJA,2000).

2.5.1. Cortes dos blocos básicos

Uma vez que os blocos básicos têm vários tamanhos, assim como diferentes números de entradas e saídas, um mecanismo fixo de hardware, tal como o Block History Buffer (BHB), não pode garantir a gravação de todos os diferentes blocos básicos. Nesta situação, por exemplo, o BHB deve descartar blocos básicos que são grandes demais para caber na entrada BHB. Por tal razão, devem-se dividir os blocos básicos em sub-blocos de tamanhos gerenciáveis, esperando melhorar o esquema de reuso de blocos.

Dividir os blocos básicos não é uma tarefa totalmente simples, uma vez que tem impactos positivos e negativos na oportunidade de reuso. Em geral, dividir um bloco básico em sub-blocos menores requer que poucas instruções tenham suas entradas repetidas simultaneamente. Algumas entradas tipicamente se traduzem em mais oportunidades de reuso, uma vez que requerem que alguns poucos valores se repitam ao mesmo tempo. Entretanto, esses blocos menores também significam saltar poucas instruções para cada bloco reusável, o que torna mais difícil minimizar o custo. O objetivo é encontrar um ponto de equilíbrio para atingir o máximo desempenho para o reuso de blocos.

Além de alterar as características de entrada, cortar os blocos básicos pode reduzir a quantidade total de localidade de valor disponível. Algumas dependências intra-blocos podem ser convertidas em dependências inter-blocos em decorrência desta

divisão, o que pode reduzir as chances de reuso de valor. Mais ainda, algumas saídas mortas são convertidas para saídas vivas, desde que o último uso do valor possa ser retardado para o próximo bloco. Esta mudança tem um impacto negativo no reuso de valor, uma vez que agora, mais valores são requeridos para repetir a mesma seqüência de instruções, para constituir a localidade de valor.

Por outro lado, o corte de blocos básicos em sub-blocos tem efeitos positivos. Por exemplo, um bloco pode ter dois grafos de dependência essencialmente separados para duas tarefas independentes. A computação para uma tarefa pode ter uma boa localidade de valor de entrada e saída, mas a computação para outra tarefa não. Combinar a computação das duas tarefas em um bloco básico reduz a possibilidade do reuso de bloco. Neste caso, dividir o bloco básico no limite da tarefa pode realmente possibilitar mais oportunidades de reuso de valor (HUANG;LILJA,2000).

2.6. Reuso em nível de traços

Este trabalho explora técnicas de hardware para o reuso de traços. Explorar o reuso em nível de traço implica no fato de que uma simples operação de reuso pode evitar a execução de um número potencialmente grande de instruções. O mais importante é que estas instruções não precisam ser buscadas, portanto não consomem largura de banda de busca. Finalmente, uma vez que estas instruções não são alocadas no Reorder Buffer, elas não ocupam espaços da janela de instrução, logo, o tamanho da janela de instrução efetiva é aumentado como conseqüência. Esta técnica pode computar de uma só vez os resultados da cadeia de dependência das instruções, o que permite ao processador exceder o limite do fluxo de dados que é inerente ao programa (GONZÁLEZ;TUBELLA;MOLINA,1999).

Um traço se refere a qualquer seqüência dinâmica de instruções. Um traço diferencia-se de um bloco básico, uma vez que os traços não são necessariamente delimitados por uma instrução de desvio. Traços podem, inclusive, conter instruções de desvio em sua seqüência de instruções. O objetivo do reuso em nível de traço é evitar a execução individual das instruções de um traço. Todas as mudanças no estado do processador que seriam produzidas por estas instruções são feitas aplicando novamente estas mudanças que foram produzidas em uma execução anterior do mesmo traço,

produzidas pelas execuções que tiveram as mesmas entradas (GONZÁLEZ;TUBELLA;MOLINA,1999).

O reuso de traços requer que o processador inclua algum tipo de memória para gravar os traços anteriores, uma regra para decidir que traços são mais valiosos para serem gravados, um mecanismo para identificar quando um próximo traço pode ser usado e um processo final de atualização do estado do processador, se o traço for reusável.

A memória de reuso de traços (RTM) é uma memória que armazena traços anteriores que são candidatos a serem reusados. Do ponto de vista do reuso, um traço é identificado pelas suas entradas e saídas, como apresenta a Figura 2.10. A entrada é definida pelo endereço inicial (PC) da instrução, o conjunto de registradores e locações de memória que estão válidas e seu conteúdo antes do traço ser executado. Uma posição de memória/registrador é válida se ela é lida antes de ser escrita.

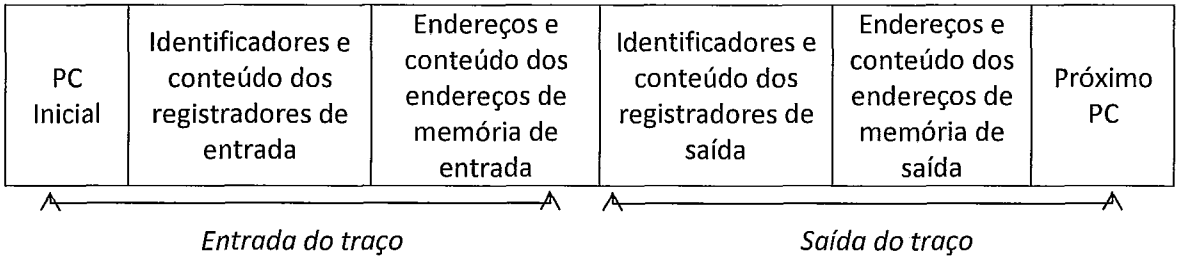


Figura 2. 10 - Estrutura de campos de uma entrada RTM

A saída do traço consiste no conjunto de registradores e locações de memória que o traço grava, seu conteúdo depois que o traço é executado e o endereço da próxima instrução que será executada após o traço.

O processador decide dinamicamente que traços do fluxo dinâmico são candidatos ao reuso. Diferentes heurísticas podem ser usadas para decidir os pontos iniciais e finais de um traço. Um critério conveniente pode ser começar um novo traço quando uma instrução reusável é encontrada e terminar o traço antes que a primeira instrução não reusável seja encontrada. Outra possibilidade que também é avaliada é considerar um tamanho fixo de traços que pode ser dinamicamente expandido uma vez que seja reusado.

Observe que traços podem ter um número variável de instruções. De fato, as instruções que fazem parte de um traço não são gravadas no RTM. Obviamente, elas são parâmetros de implementação que limitam o tamanho de um traço, tais como a quantidade de valores de entrada e saída que pode ser armazenada em cada entrada RTM, porém o número de instruções de um traço por si só não é uma limitação (GONZÁLEZ;TUBELLA;MOLINA,1999).

Em alguns pontos da execução, como na busca da instrução inicial do traço ou em qualquer momento em que operando de entrada do traço ficar pronto, por exemplo, o processador checa se o traço atual pode ser reusado. A Figura 2.11 ilustra este cenário. Em caso positivo, o processador usa a informação de traço obtida do RTM, para atualizar seu estado da seguinte forma:

a) O PC é atualizado com o campo ***nextpc*** e a unidade de busca prossegue com as instruções seguintes ao traço. As instruções que pertencem ao traço não precisam ser buscadas
b) Os registradores de saída e as locações de memória de saída são atualizados com os valores da entrada RTM

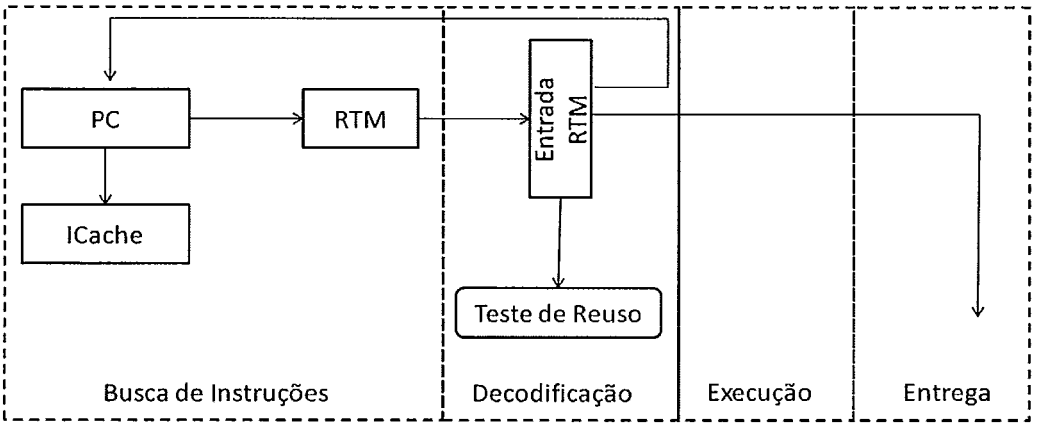


Figura 2. 11 – Reuso de traços durante as fases do pipeline

Existem basicamente duas abordagens para identificar se o traço é reusável. Uma possibilidade é ler os valores atuais de todos os registradores e locações de memória de entrada e compará-los com os valores de qualquer entrada RTM associada ao PC atual. Outra possibilidade é somar a cada entrada RTM um bit válido. Quando um traço é gravado, seu bit válido é ajustado. Para gravação de registrador ou memória, todas as entradas RTM com uma posição correspondente e toda a sua lista de entrada

são invalidadas. A última abordagem requer um teste mais simples de reuso, que consiste apenas em checar o bit de validade (GONZÁLEZ;TUBELLA;MOLINA,1999).

2.6.1. O potencial de desempenho do reuso em nível de instruções e traços

O cenário analisado é focado numa janela de instrução limitada, porém com um número infinito de unidades funcionais. Deste modo, não se considera o benefício de se reduzir a contenção de unidades funcionais, que devido ao aumento contínuo de transistores por chip terá um impacto baixo nos processadores de alta performance futuros. Mais ainda, quando o número de unidades funcionais for um gargalo, aumentar o número de unidades funcionais será mais custo-efetivo que implementar um esquema de reuso. Também se considera o caso da janela de instrução infinita, como uma indicação dos limites do potencial dessas técnicas.

Para um cenário de janela de instrução infinita, o tempo de execução só é limitado pela dependência de dados entre instruções, tanto através de registrador quanto de memória. Para o cenário de janela limitada, a ordem de execução de W instruções, é apenas limitada pela dependência de dados, onde qualquer par de instruções a uma distância maior que W deve ser executado seqüencialmente.

O IPC para uma máquina de janela infinita é computado analisando o fluxo dinâmico de instruções. Para cada instrução, seu tempo de conclusão é determinado como o máximo do tempo de conclusão dos produtores de todas as entradas, somado à sua latência. As entradas de uma instrução podem ser registradores ou operandos de memória. Então, para cada registrador lógico e cada posição de memória, o tempo de conclusão da ultima instrução que foi atualizada é mantido na tabela. Uma vez que todo o fluxo de instruções dinâmicas foi processado, o IPC é computado como o resultado da divisão do número de instruções dinâmicas, pelo tempo máximo de conclusão de cada instrução.

O processo de computação do PC para o cenário de janela de instrução limitada é uma extensão da abordagem da janela ilimitada. A extensão consiste em computar o tempo de graduação de cada instrução, como o máximo do tempo de conclusão de qualquer instrução prévia, incluindo ela mesma. Então, o tempo de término de execução de uma instrução dada é computado como o máximo dentre os tempos de conclusão de

todos os produtores de suas entradas e o tempo de graduação das instruções em locações W acima do traço, mais a latência da instrução.

Para a análise de desempenho do reuso de valor, primeiro foi considerado um componente de reuso com infinitas tabelas para manter o histórico de instruções/traços prévios e analisamos o efeito de diversas latências de reuso. A latência de reuso corresponde ao tempo que a alteração de reuso consome. O teste de reuso é baseado em uma busca associativa de traços que começam no mesmo PC (GONZÁLEZ;TUBELLA;MOLINA,1999).

2.7. Reuso de Load

As técnicas de reuso de Load têm por objetivo a manutenção de um histórico de execução de instruções de leitura da memória. Este histórico é utilizado para detectar se uma execução subseqüente de determinada instrução produzirá os mesmos resultados de uma instrução anterior. Em caso positivo, estes resultados são disponibilizados para as instruções que dependem destes, sem que precisem ser executadas (YANG;GUPTA,2000).

Yang e Gupta (2000) propõem um mecanismo em hardware para implementar o Load Reuse. São consideradas as oportunidades de reuso em duas situações: execuções prévias de uma mesma instrução load ou execuções prévias de instruções loads diferentes.

O mecanismo de Load Reuse que explora estes tipos de redundância consiste em duas etapas: a etapa de ligação e a etapa de exploração. A etapa de ligação realiza a detecção de oportunidades de reuso que se encontram em instruções loads diferentes que foram executadas e faz a ligação entre este load e outras instruções load, cujo reuso de load foi detectado. Já a etapa de exploração utiliza um histórico de execuções prévias do próprio load, assim como de outras instruções loads que estão ligadas ao load corrente, a fim de detectar e explorar a redundância deste load ou de diferentes loads que ainda serão executados. Uma vez sendo detectada a redundância, a execução deste load não é realizada. Para a realização das etapas mencionadas são necessárias duas estruturas de hardware, uma para cada etapa do algoritmo (YANG;GUPTA,2000).

2.7.1. A estrutura de ligação

Para detectar a existência do reuso em loads diferentes, esta estrutura de detecção é usada para manter informações sobre as instruções load que foram executadas recentemente. Ao chegar o término da execução de uma instrução load, seu resultado é comparado com a informação que já está gravada, visando verificar se existe possibilidade de reuso entre este load e outros loads recentemente executados, limitando a quantidade de informação gravada e garantindo que a maioria das oportunidades de reuso seja detectada (YANG;GUPTA,2000).

2.7.2. A estrutura de exploração

Quando uma instrução load é executada, todo o histórico de sua execução é armazenado em uma Load Table. Esta tabela é utilizada para evitar execuções de futuros loads, caso seja detectada a presença de redundância. Quando uma nova instrução load é encontrada, seus operandos são comparados com os operandos dos loads selecionados na Load Table. Estes loads selecionados correspondem às execuções de loads anteriores que foram ligados ao load atual, na etapa de ligação, em decorrência da redundância entre loads diferentes. Se a comparação entre estes operandos resultar em sucesso e o valor do load ainda estiver válido, o valor pode ser reusado a partir da tabela e este load precisará fazer acessos à memória.

Para detectar o reuso a partir de diferentes stores, um histórico de execuções destas instruções também deve ser mantido. Uma possibilidade é gravar as informações de loads e stores recentes em uma mesma fila, porém não seria a melhor opção, uma vez que esta poderia ser toda preenchida com instruções store e, conseqüentemente, oportunidades de reuso para diferentes loads não seriam detectadas. Deste modo, o histórico de execuções recentes de stores é mantido em uma fila separada, a Recent Store Queue (RSQ) (YANG;GUPTA,2000).

2.7.3. O hardware

Para a implementação deste esquema, existem dois componentes responsáveis pelo reuso. Um dos componentes é responsável pela exploração de oportunidades de reuso entre loads, sejam eles iguais ou diferentes, e o outro componente é responsável pelo reuso dos stores.

A Load Table (LT) é utilizada para armazenar o histórico dos loads encontrados durante a execução. Nesta tabela é alocado no máximo um registro por load e a tabela é indexada a partir do PC de cada instrução. Um histórico de execução consiste de operandos dinâmicos, de onde o endereço é computado e o valor é carregado. A Load Table é mostrada na Figura 2.12. Além disso, existe um bit ***mem***, que indica se o valor carregado é válido ou se foi invalidado por alguma instrução store que realizou uma gravação no mesmo endereço. O bit ***mem*** deve ser invalidado sempre que um store realiza uma gravação em algum endereço encontrado na tabela. Neste momento, o valor que é gravado pela instrução store ainda não é conhecido (YANG;GUPTA,2000).

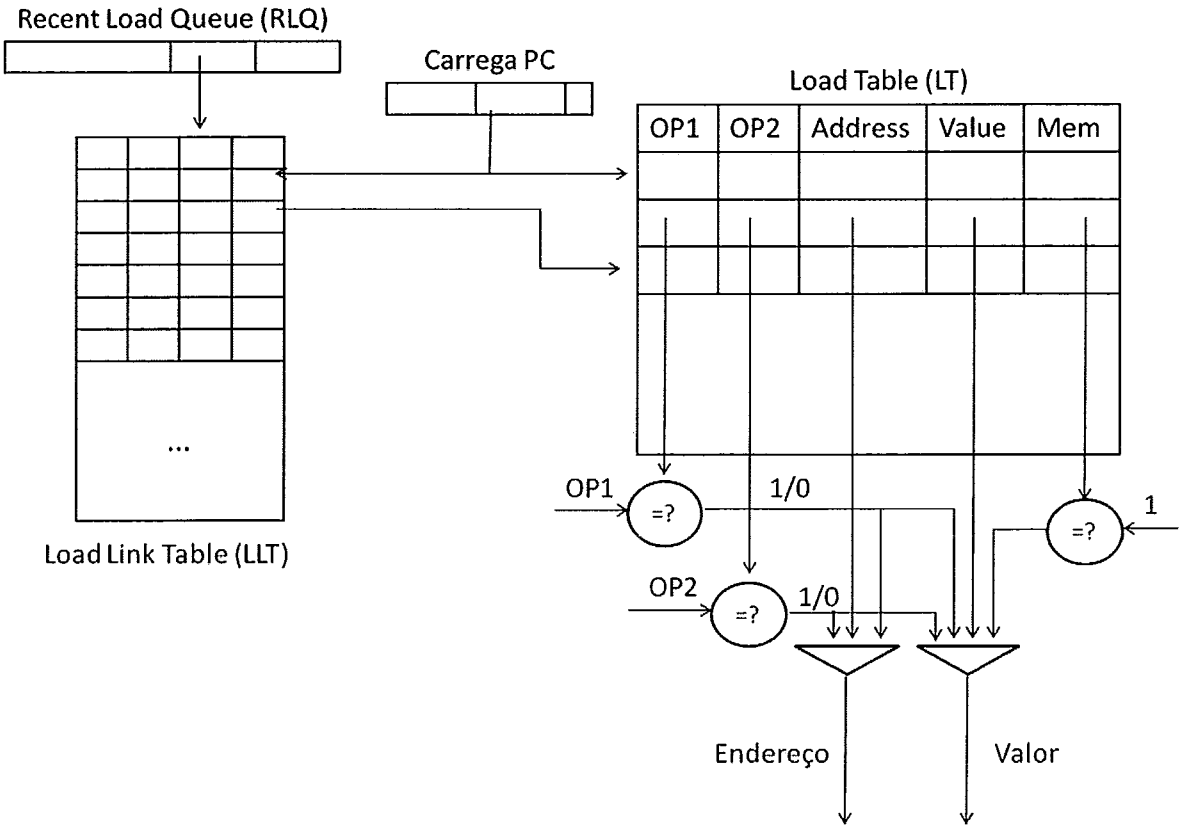


Figura 2. 12 - Estrutura da Load Table

Quando uma instrução load é encontrada durante a execução, a LT é consultada, visando localizar uma entrada correspondente na LT, que possibilite o reuso de mesmo load. Os operandos de entrada desta instrução são comparados com os que estão armazenados na entrada da LT. Se os operandos forem iguais e o bit ***mem*** for válido, o endereço de load e seu valor podem ser carregados diretamente da LT, sem ser necessário calcular o endereço novamente, nem carregar o valor da memória, o que significa que a execução deste load pode ser completada. Se os operandos são os mesmos, mas o bit ***mem*** for inválido, o endereço pode ser reusado e o valor não, devendo o mesmo ser carregado da memória.

A Load Linking Table (LLT) é utilizada para fazer a ligação de um load com loads anteriores, quando é detectada a oportunidade de reuso entre loads diferentes. A LLT é indexada através do PC da instrução load. Cada entrada da LLT consiste no conjunto de reuso deste load. Deste modo, a LT indexa até 4 loads para uma entrada LLT. O histórico dos loads é obtido na LT e os operandos também são comparados com os operandos do load, encontrados durante a execução.

A Recent Load Queue (RLQ) tem como função detectar as oportunidades de reuso de diferentes loads em execuções anteriores e incluir os registros na LLT, a fim de que possam ser explorados posteriormente. A RLQ é uma FIFO que contém o histórico de um número pré-estabelecido de loads mais recentemente executados. Quando um load que pode ser reusado completa a sua execução, o histórico desta instrução é comparado com outros loads na RLQ. Se um acerto ocorre, o índice da LT que corresponde ao registro reusado é inserido no conjunto de reuso do load executado (YANG;GUPTA,2000).

Se um load que está sendo executado não puder reusar um load anterior, seja diretamente através da LT ou indiretamente, através da LLT, há a tentativa de utilizar o endereço e o valor, a partir de um store recentemente executado. Para que tal tentativa seja possível, existe uma estrutura FIFO similar à RLQ: a Recent Store Queue (RSQ). A diferença entre a store queue padrão e a RSQ é que esta última contém os operandos a partir de onde o endereço é calculado, permitindo não só o reuso do valor, mas também o cálculo do endereço. O bit ***mem*** é mantido, então apenas o último valor do store em um determinado endereço pode ser reusado. À medida que novos stores são encontrados, instruções stores anteriores que gravaram no mesmo endereço são invalidadas na RSQ. A estrutura da RSQ é ilustrada na Figura 2.13.

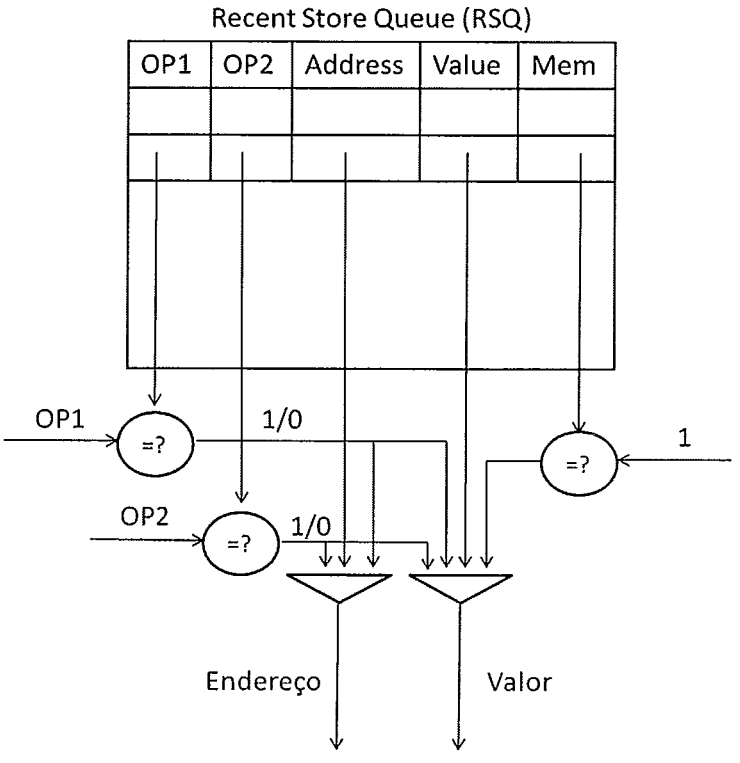


Figura 2. 13 – Estrutura da Recent Store Queue

3. Mecanismos de Reuso Dinâmico de Traços: DTM e RST

Neste capítulo serão apresentados os trabalhos na área de reuso de computação, com granularidade em nível de traços, que serviram de base para este trabalho de Dissertação: o DTM - Dynamic Trace Memoization (COSTA,2001) e o RST - Reuse Through Speculation on Traces (PILLA,2004), que é uma evolução do DTM, que agrega a previsão de valores ao reuso de traços já implementado pelo trabalho anterior.

3.1. DTM: *Dynamic Trace Memoization*

Neste trabalho, foi estudado o reuso dinâmico de traços, usando a técnica de memoização (COSTA;FRANÇA;CHAVES FILHO,2000). Esta técnica objetiva a aceleração das execuções, armazenando os resultados de uma determinada função, para reuso posterior, ao invés de recomputá-los a cada nova chamada de função.

Um traço é uma seqüência dinâmica de instruções emitidas durante a execução de um programa. Um traço redundante é composto apenas de instruções redundantes. Entende-se por instrução redundante toda instrução que é passível de ser reusada, por possuir o mesmo contexto de entrada por produzir as mesmas saídas que outras instâncias de uma mesma instrução ou até mesmo de instruções diferentes. Ao contrário de outros esquemas de reuso, traços redundantes no DTM não incluem instruções load e store. Entretanto, o DTM pode reusar os cálculos de endereço associados às instruções de acesso a memória.

Os aspectos a serem analisados durante a operação do DTM são: como instruções redundantes são detectadas; como as instruções redundantes são utilizadas para construir traços e como traços e instruções redundantes são identificados e reusados (COSTA,2001).

A detecção de instruções redundantes é o primeiro passo necessário tanto na construção do traço quanto no reuso de uma única instrução. Isto envolve verificar se a instância dinâmica atual de uma instrução estática utiliza os mesmos valores de

operandos de uma instância anterior. Para tal verificação, o DTM emprega uma tabela totalmente associativa chamada Memo_Table_G.

Cada entrada da Memo_Table_G mantém informações associadas à instrução dinâmica. A Figura 3.1 (COSTA,2001) exibe a estrutura de uma entrada desta tabela. O campo ***pc*** contém o endereço estático da instrução. Dois campos, ***sv1*** e ***sv2***, mantêm os valores dos operandos usados pela instrução dinâmica. Um campo **res/targ** grava tanto o resultado de uma instrução lógico-aritmética, quanto o endereço alvo de uma instrução de transferência de controle. Dois bits, ***jmp*** e ***brc***, identificam a instrução como um salto ou um desvio, respectivamente. Outro bit ***btaken*** indica se o desvio será tomado ou não. A Memo_Table_G não mantém código de instruções, e sim o endereço da instrução (PC), que é utilizado para indexar esta tabela de reuso (COSTA,2001).

Cada instrução dinâmica é classificada como redundante ou não redundante. Instruções de acesso à memória como load e store são marcadas como não-redundantes. O endereço e os valores dos operandos atuais de uma instrução dinâmica válida são comparados com os campos **pc**, **sv1** e ***sv2*** das entradas da Memo_Table_G. Se não ocorrer uma correspondência, a instrução é marcada como não-redundante. Uma entrada é alocada na Memo_Table_G e os campos ***pc***, ***sv1*** e ***sv2*** são preenchidos apropriadamente. Se um acerto acontece, a instrução é marcada como redundante, porém não é alocada uma entrada na Memo_Table_G. Se a instrução dinâmica é não redundante, a construção do traço é finalizada. Se a instrução é redundante, os contextos de entrada e saída do traço atualmente em construção são atualizados.

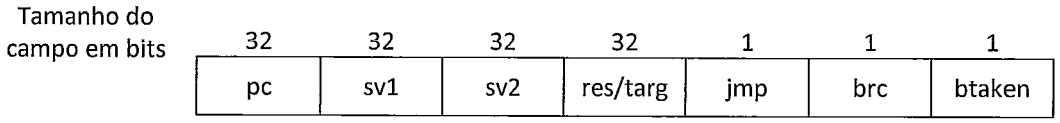


Figura 3. 1 - Estrutura de uma entrada da tabela Memo_Table_G

O contexto de entrada de um traço é definido como o conjunto de valores dos operandos usados pelas instruções que estão formando o traço, porém que foram produzidas por instruções fora do traço. O contexto de saída é o conjunto de resultados produzidos pelas instruções dentro do traço. A construção do traço consiste na inclusão de valores ao contexto de entrada e saída do traço para cada instrução redundante encontrada (COSTA,2001).

O mapa do contexto de entrada e o mapa do contexto de saída têm um bit para cada registrador arquitetural. Para cada instrução redundante, um bit do mapa do contexto de entrada é ativado, caso o registrador de origem correspondente mantenha um valor que foi escrito anteriormente por uma instrução fora do traço que está sendo construído. O bit do mapa de contexto de saída que corresponde ao registrador de destino da instrução também é ativado. Em qualquer lugar que o bit do mapa de contexto de entrada/saída seja ativado, o conteúdo do respectivo registrador é adicionado ao contexto de entrada/saída do traço em construção.

Na Figura 3.2, é mostrada a estrutura de armazenamento de traços que é requerida para a implementação do DTM. Os buffers Buffer_T#1 e Buffer_T#2 mantêm informações sobre o traço em construção. Eles permitem que a construção de um novo traço comece antes mesmo que a informação do traço anterior seja salva. Uma vez que a construção do traço é encerrada, a informação do traço é transferida para uma tabela chamada Memo_Table_T.

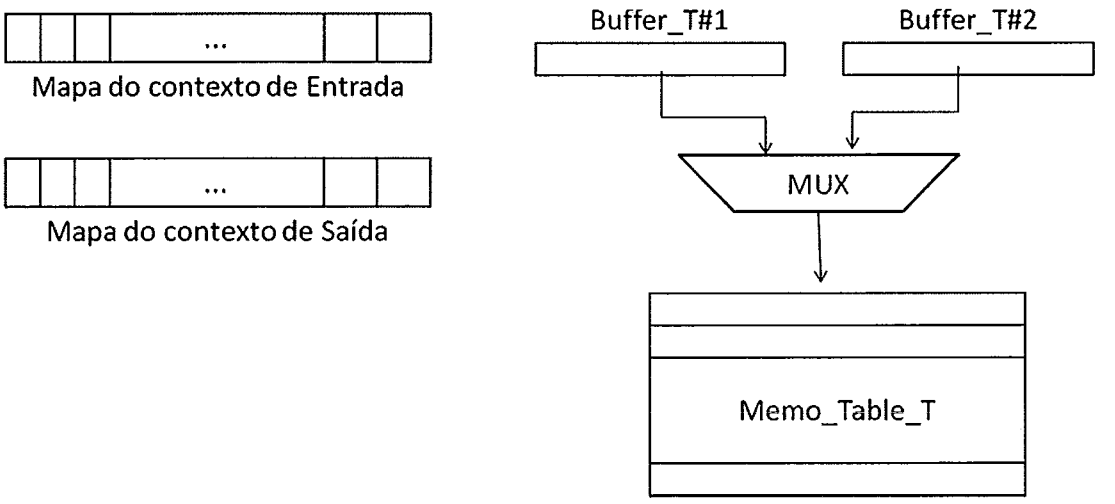


Figura 3. 2 – Estruturas para o armazenamento de traços no DTM

O campo ***pc*** guarda o endereço da instrução inicial, enquanto o campo ***npc*** especifica o endereço da próxima instrução a ser executada no caso do traço ser reusado. O campo ***icv*** guarda os valores do contexto de entrada, enquanto o ***icr*** guarda os índices dos registradores de origem correspondentes. O campo ***ocv*** guarda os valores do contexto de saída, enquanto os registradores de destino são salvos no campo ***ocr***.

Cada identificador de registrador tem um bit de validade. Os bits no campo ***bmask*** indicam um desvio dentro do traço e o bit correspondente ***btaken*** indica de esse desvio é tomado ou não.

É importante observar que Memo_Table_T também não grava código de instruções e mantém múltiplas instâncias do mesmo traço, onde cada instância possui diferentes contextos de entrada. O ***context size*** N especifica o tamanho máximo de elementos no contexto de entrada e saída. O ***branch limit*** B indica o número máximo de desvios dentro do traço (COSTA,2001).

A construção de um traço é finalizada se:

- uma instrução não-redundante é encontrada;
- se o número de elementos do contexto atingiu o limite N;
- o número de desvios dentro do traço atingiu o limite B.

A Memo_Table_T é empregada para detectar a redundância de instâncias de traços. O formato das entradas de Memo_Table_T é apresentado na Figura 3.3 (COSTA,2001). O endereço de cada instrução dinâmica é comparado com os campos ***pc*** nas entradas Memo_Table_T. Para as entradas que correspondem aos campos ***pc***, os conteúdos atuais dos registradores apontados pelos campos ***icr*** válidos são comparados com os valores gravados nos campos ***icv***. O traço é redundante se os valores dos registradores correntes correspondem ao contexto de entrada de uma instância prévia do traço gravado em Memo_Table_T.

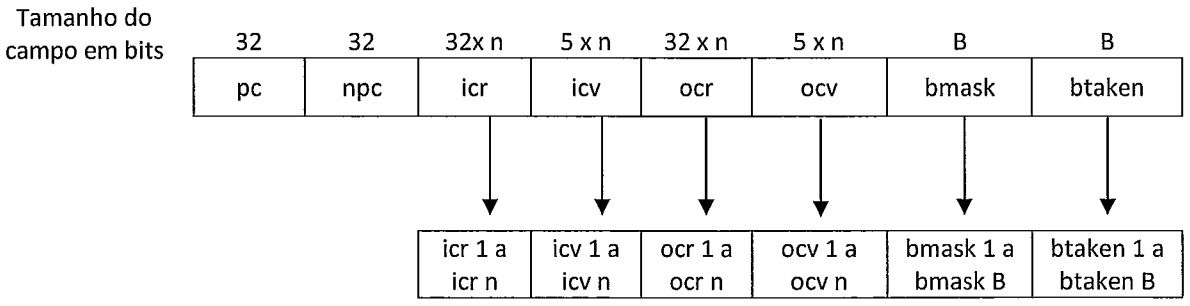


Figura 3. 3 – Estrutura de uma entrada da tabela Memo_Table_T

Em microarquiteturas que fazem busca de múltiplas instruções, diversos endereços de instruções podem corresponder a várias entradas simultaneamente nos campos **pc** em Memo_Table_T. Isto significa que instâncias de diferentes traços são candidatos a serem redundantes, uma vez que cada endereço distinto representa um traço diferente. Por esta razão, o teste de redundância é executado apenas nas instâncias do mesmo traço.

A Memo_Table_G é empregada para determinar se uma dada instrução dinâmica é redundante ou não. Dizemos que um acerto na Memo_Table_G ocorre onde quer que uma instrução redundante seja detectada. A Memo_Table_T é empregada para detectar instâncias redundantes de traços. Um acerto em Memo_Table_T ocorre se uma instância de um traço redundante é encontrada.

A decisão de reusar uma instrução ou um traço é baseada na ocorrência de acertos na Memo_Table_G e/ou na Memo_Table_T. Uma instrução é reusada se ocorre um acerto apenas na Memo_Table_G. Neste caso, o resultado da instrução é obtido do campo ***res/targ*** na entrada correspondente da Memo_Table_G para a instrução redundante. Um traço é reusado toda vez que um acerto ocorre em Memo_Table_T, mesmo que o acerto ocorra nas duas tabelas. Os valores do contexto de saída são gravados nos registradores apropriados. Além disso, o endereço do campo ***npc*** é carregado no PC e o estado da predição de desvio é atualizado no campo ***btaken***.

3.1.1. A microarquitetura com DTM

O estágio de Busca de Operandos lê os valores válidos dos operandos do arquivo de registradores e insere instruções com seus operandos no topo do Buffer de Reordenação. O Buffer de Emissão provê as funcionalidades de Reorder Buffer e estações de reserva de forma centralizada. Cada entrada do Buffer de Emissão também faz a renomeação implícita de registradores, logo não há distinção entre registradores físicos e arquiteturais. O Estágio de Emissão seleciona as instruções do Buffer de Emissão para execução, de acordo com a disponibilidade dos operandos. As instruções têm sua execução concluída no fim do Estágio de Emissão.

O mecanismo DTM possui estágios de pipeline: DS1, DS2 e DS3, que operam em paralelo com os estágios Busca de Instruções, Busca de Operandos e Estágio de Entrega do pipeline tradicional. A Figura 3.4 ilustra a microarquitetura do DTM com seus principais componentes. Os itens seguintes apresentam um detalhamento da microarquitetura DTM ilustrada.

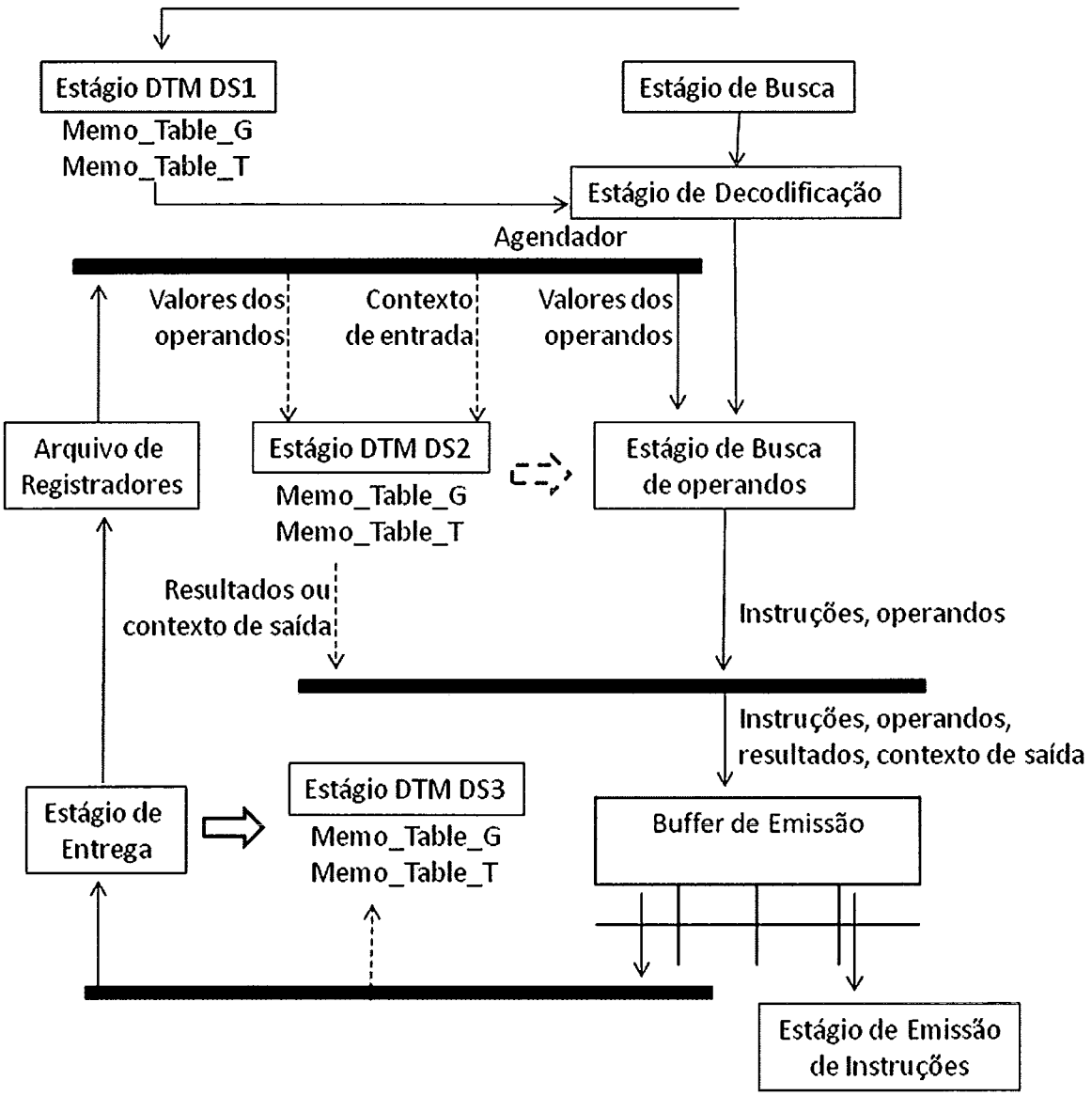


Figura 3. 4 – Microarquitetura do DTM

- **Inclusão de instruções na Memo_Table_G**

O DS1 aloca uma entrada Memo_Table_G, para cada instrução válida buscada. O DS1 fornece os índices das entradas mais recentemente alocadas para o Estágio de Decodificação, que anexa cada índice à respectiva instrução antes do despacho. O DS3 verifica o endereço, valor do operando e resultado de cada instrução lida pelo Estágio de Entrega e preenche a entrada Memo_Table_G apontada pelo índice anexado à instrução. Se a instrução for descartada pelo Estágio de Entrega, o DS3 libera a entrada Memo_Table_G. O DTM captura apenas instruções ao longo de caminhos de execução corretos.

- **Identificação de instruções redundantes**

O DS1 compara o endereço de cada instrução buscada com os campos ***pc*** da Memo_Table_G e seleciona aquelas entradas cujo PC corresponde a um acerto na busca. O DS2 verifica os valores válidos dos operandos lidos pelo Estágio de Busca de Operandos e os compara com o conteúdo dos campos ***sv1*** e ***sv2*** nas entradas de Memo_Table_G pré-selecionadas por DS1. Se um acerto ocorre, o DS2 indica que o Estágio de Busca de Operando na instrução correspondente é redundante.

- **Atualização do contexto do traço**

O DS3 verifica a identificação do registrador de origem, os valores dos operandos, a identificação do registrador de destino e o resultado de cada instrução concluída. Se a instrução é redundante, o DS3 atualiza os mapas de contexto e preenche os campos apropriados em buffers temporários. Se a instrução não é redundante, o DS3 finaliza a construção do traço, salvando a informação em uma entrada da Memo_Table_T.

- **Seleção das instruções e traços redundantes candidatos**

O DS1 compara cada endereço buscado com os campos PC da Memo_Table_G e Memo_Table_T e seleciona as entradas onde o PC corresponde a um acerto. O DS1 seleciona apenas as entradas em Memo_Table_T que guardam instâncias do mesmo traço.

- **Identificação de uma instrução ou traço redundante**

O DS2 procura instruções redundantes na Memo_Table_G. Além disso, este estágio lê os registradores marcados como válidos em campos ***icr*** nas entradas previamente selecionadas de Memo_Table_T e compara os valores dos operandos com o conteúdo de ***icv***.

- **Reuso de uma instrução ou traço**

Se um acerto ocorre em Memo_Table_G, o Estágio de Busca de Operando despacha a instrução redundante e seus resultados, posteriormente obtidos em Memo_Table_G, para o Buffer de Emissão. Se um acerto em Memo_Table_T ocorre, o Estágio de Busca de Operando despacha até N (ocr, ocv) pares no Buffer de Emissão. Para cada caso, as entradas envolvidas do Buffer de Emissão não são consideradas para o agendamento, mas os resultados que eles contêm podem ser encaminhados para outras instruções.

3.1.2. Implementação do Mecanismo DTM

O caminho crítico do mecanismo DTM é compreendido pelas operações envolvidas no reuso de traço:

(1) Leitura dos valores do contexto de entrada atual do arquivo de registradores

(2) Comparação com os valores do contexto de entrada anterior gravado em Memo_Table_T

(3) Gravação dos valores do contexto de saída no Buffer de Emissão

A operação (1) emprega portas dedicadas do arquivo de registradores, deste modo pode executar em paralelo a leitura do operando com o Estágio de Busca de Operando. A operação (3) apenas substitui o despacho de instruções tradicional. Portanto, as operações DTM (1) e (3) são sobrepostas por outras operações já executadas na microarquitetura substrato.

A operação (2) adiciona um retardo ao tempo de ciclo de clock. O retardo total pode ser parcialmente oculto pelo tempo de acesso da instrução ao cache e o atraso ainda notado pode ser aceito, dependendo do ciclo de clock desejado. Entretanto, para ciclos de clock muito pequenos, próximos ou superiores a 1GHz, um pipeline extra é necessário para acessos a Memo_Table_T.

Outra questão analisada neste trabalho é o número de portas de leitura do arquivo de registradores, requerida pelo mecanismo DTM. O estágio DS2 não requer portas de leitura extras para acessar os operandos correntes de instruções individuais. Este estágio apenas verifica o operando, normalmente lido no Estágio da Busca de Operando. Entretanto, o DTM requer portas de leitura adicionais, a fim de permitir que valores do contexto de entrada sejam lidos concorrentemente com a busca de operandos do pipeline tradicional (COSTA,2001).

A principal diferença do DTM, com relação aos outros esquemas de reuso mencionados é que, até que uma instrução não redundante seja encontrada, não existem limites para o reuso, como ocorre com o reuso de blocos básicos ou de instruções isoladas. Neste trabalho, instruções de acesso à memória não foram consideradas, para evitar incorrer em penalidades de tempo e custo relacionadas a manter tabelas de memória consistentes, no que concerne à memória de dados.

3.2. RST: *Reuse Through Speculation on Traces*

Baseado no trabalho de DTM (COSTA,2001), foi verificado que o reuso de traços poderia ser potencializado com a agregação da predição de valores ao reuso de traços viabilizado na técnica do DTM. Com o RST é possível reusar traços cujos operandos ainda não possuam seus resultados prontos, quando a execução alcança o início do traço. Isto é possível através da integração do reuso de instruções, traços e predição de valores (PILLA,2004).

Analisou-se, através de simulações, usando os benchmarks do SPECInt95 e SPECInt2000 que cerca de 68% dos traços passíveis de reuso não foram reusados devido à indisponibilidade de alguns operandos no contexto de entrada do traço, no momento do teste de reuso. Concluiu-se então que a situação se agravaria em situações onde pipelines mais profundos fossem considerados, uma vez que os traços demandariam mais ciclos para serem criados, mais entradas estariam indisponíveis no teste de reuso, visto que as instruções consumiriam mais ciclos para serem executadas (PILLA,2004).

3.2.1. O RST e sua integração com o DTM

O RST é implementado como o DTM, quando todos os operandos estão prontos para reuso do traço e são os mesmos armazenados no contexto de entrada do traço, porém pode trabalhar de forma especulativa, quando existem valores desconhecidos no contexto de entrada do traço. Desta forma, os traços que não foram reusados através da técnica do DTM também são reusados, porém explorando não só sua redundância, como também seu potencial de predição (PILLA,2004).

Para a implementação do RST, tendo a arquitetura DTM como substrato, não é necessário nenhuma tabela extra para armazenar os valores que serão preditos. Os valores do contexto de entrada dos traços, já armazenados na tabela de memorização dos traços do DTM são os mesmos a serem utilizados na predição de valores.

Um inconveniente da predição de valores é uma demanda maior de recursos. No momento em que um valor é predito, a disponibilização de mais instruções pode consumir mais unidades funcionais, além da largura de banda de emissão e despacho. Quando a predição é incorreta, os recursos que são consumidos nesta execução incorreta ainda impedem que novas instruções que não serão executadas em modo especulativo possam ser executadas (PILLA,2004).

Quando traços são reusados de forma especulativa no RST, os valores de saída são enviados diretamente para o estágio de entrega, onde as instruções aguardam por estes valores. As fases de despacho, emissão e execução são ignoradas para todo o traço e são realizadas em um único ciclo, uma vez que foi determinado que este traço é reusável. Por esta razão, o reuso especulativo reduz a demanda pelos recursos disponíveis em muitos casos. Mesmo que uma predição seja incorreta, muitas instruções

podem ser reusadas, ao invés de serem novamente executadas, o que mais uma vez reduz a sobrecarga de execução.

O RST não aumenta os acessos às Tabelas de Memorização ou ao Arquivo de Registradores. Os acessos já são feitos nas técnicas de reuso do DTM. A diferença é que estes são inúteis se algumas das entradas não estiverem prontas para serem testadas.

O RST pode fazer reuso tanto de instruções como de traços, porém apenas traços são reusados em modo especulativo, porque mais de uma instrução pode ser encapsulada, reduzindo a complexidade do número de testes que serão necessários no estágio de entrega (PILLA,2004).

3.2.2. Reuso e construção de traços no RST

A construção de traços dinamicamente, bem como a inclusão das instruções e traços redundantes nas Tabelas de Memorização são realizadas da mesma forma que no DTM, porém o RST também inclui instruções que não estão na Memo_Table_G, mas que fazem parte do domínio de reuso. Desta forma, mais traços podem ser criados e reusados. A desvantagem desta abordagem é que as instruções que foram encontradas em Memo_Table_G são mais propensas a serem encontradas novamente com as mesmas entradas, portanto em alguns casos, alguns traços podem ser eliminados antes mesmo de serem reusados.

Existem duas formas de construção de traços. O modo reusável, quando instruções que não estão na Memo_Table_G são inseridas em um traço e o modo somente-reusadas, quando apenas instruções pertencentes à Memo_Table_G são permitidas em um traço. É possível alternar entre os modos, adaptando à arquitetura às particularidades de um determinado programa em execução.

Pela política do RST, o processo de criação de um traço permite a inclusão de instruções que ainda não foram reusadas (não possuem entradas na Memo_Table_G) em um traço que está sendo formado. As instruções vão sendo adicionadas até que uma instrução que não pertence ao domínio de instruções reusáveis seja encontrada. Neste caso, a instrução não reusável encerra a construção do traço corrente. Este traço é então incluído na Memo_Table_T.

As principais diferenças entre a política do DTM são:

- traços são formados mais rapidamente no RST, uma vez que as instruções não necessitam ser marcadas como redundantes para serem incluídas em um traço, por isso não há necessidade de inclusão prévia da instrução em Memo_Table_G;
- uma maior quantidade de traços é criada no RST, conseqüentemente existe uma quantidade potencialmente maior de instruções candidatas ao reuso;
- traços são menos redundantes que no DTM, uma vez que as instruções que são adicionadas aos traços não necessitam ser redundantes; em compensação, instruções redundantes são incluídas antecipadamente no traço e por isso o RST pode se beneficiar de traços que não seriam formados no DTM;
- Memo_Table_T é acessada mais vezes para gravar traços e em decorrência disto, pode incluir registros com muito pouca redundância.

3.2.3. Teste de predição incorreta e recuperação

Quando um traço é reusado especulativamente, o valor predito e a informação de quem são os produtores dos valores preditos são gravados. Quando os produtores terminam sua execução, os valores preditos e as atuais saídas são comparados. Se estes corresponderem, o estágio de entrega é informado. As saídas do traço reusado de modo especulativo podem ser gravadas, assim como as instruções subseqüentes ao traço que estão prontas podem ser entregues. Se qualquer um dos testes de valor não corresponder, um mecanismo de recuperação é ativado, do mesmo modo que em predições de desvio. As instruções posteriores ao traço e suas saídas são descartadas; a busca de instruções é redirecionada.

3.2.4. A arquitetura RST: Diferenças entre o pipeline RST e o pipeline DTM

O pipeline RST foi construído sobre a arquitetura DTM e os estágios do pipeline RST ocorrem em paralelo com o pipeline pricipal, tal qual ocorre com os estágios do DTM.

Para o pipeline RST ocorreram modificações, principalmente nos estágios RS2 e RS4, além da inclusão de um estágio intermediário, que ocorre paralelamente ao estágio de entrega do pipeline principal.

A fase DS1 possui as mesmas funcionalidades que RS1, assim como os estágios RS4 e DS3. O estágio RS2 possui algumas alterações no processo de teste de reuso com relação ao estágio DS2. No pipeline RST, a principal modificação está na inclusão da fase RS3, que realiza os testes de predições incorretas. Uma vez que o DTM não trabalha com reuso especulativo, o mesmo não possui esta funcionalidade em seu pipeline. Para melhor ilustrar essas diferenças arquiteturais, as Figuras 3.5 e 3.6 (PILLA,2004) mostram os estágios dos pipelines DTM e RST, respectivamente.

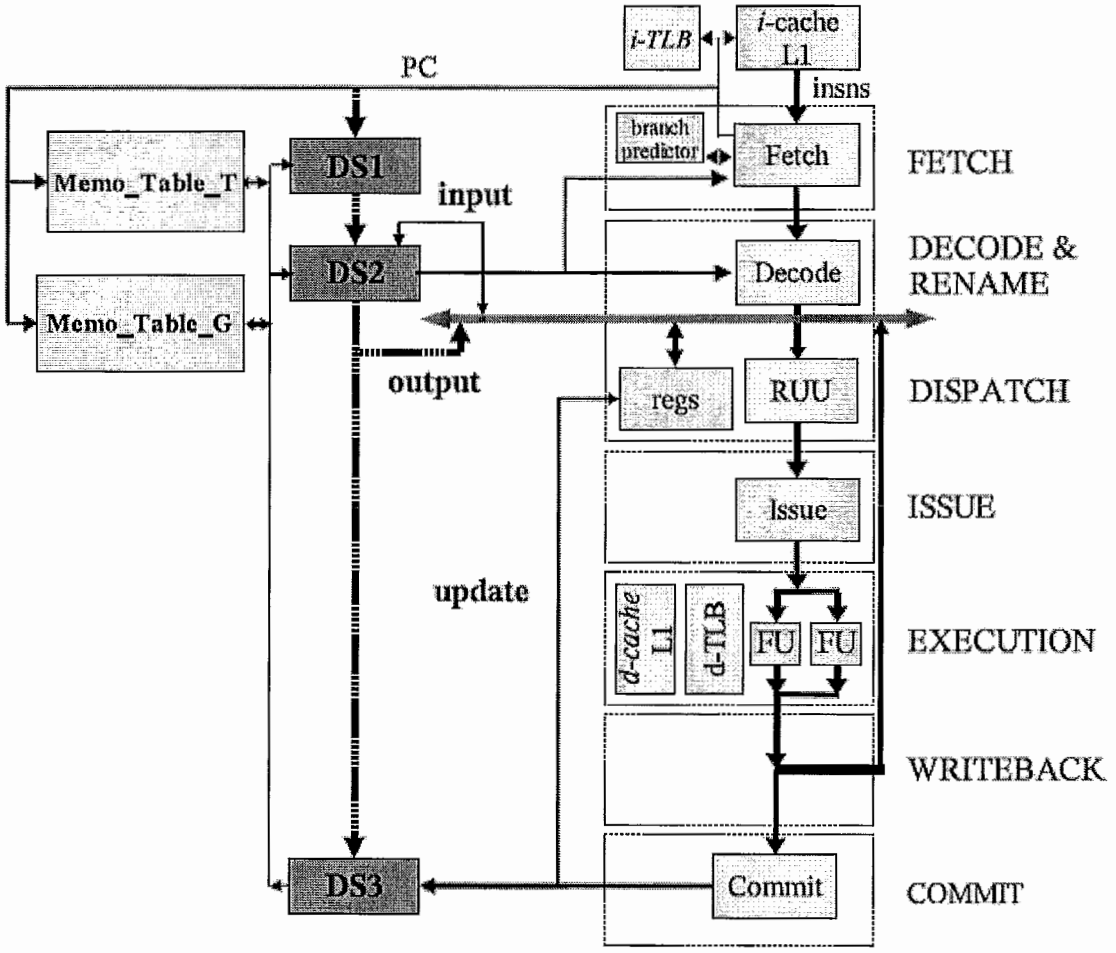


Figura 3. 5 – Pipeline DTM

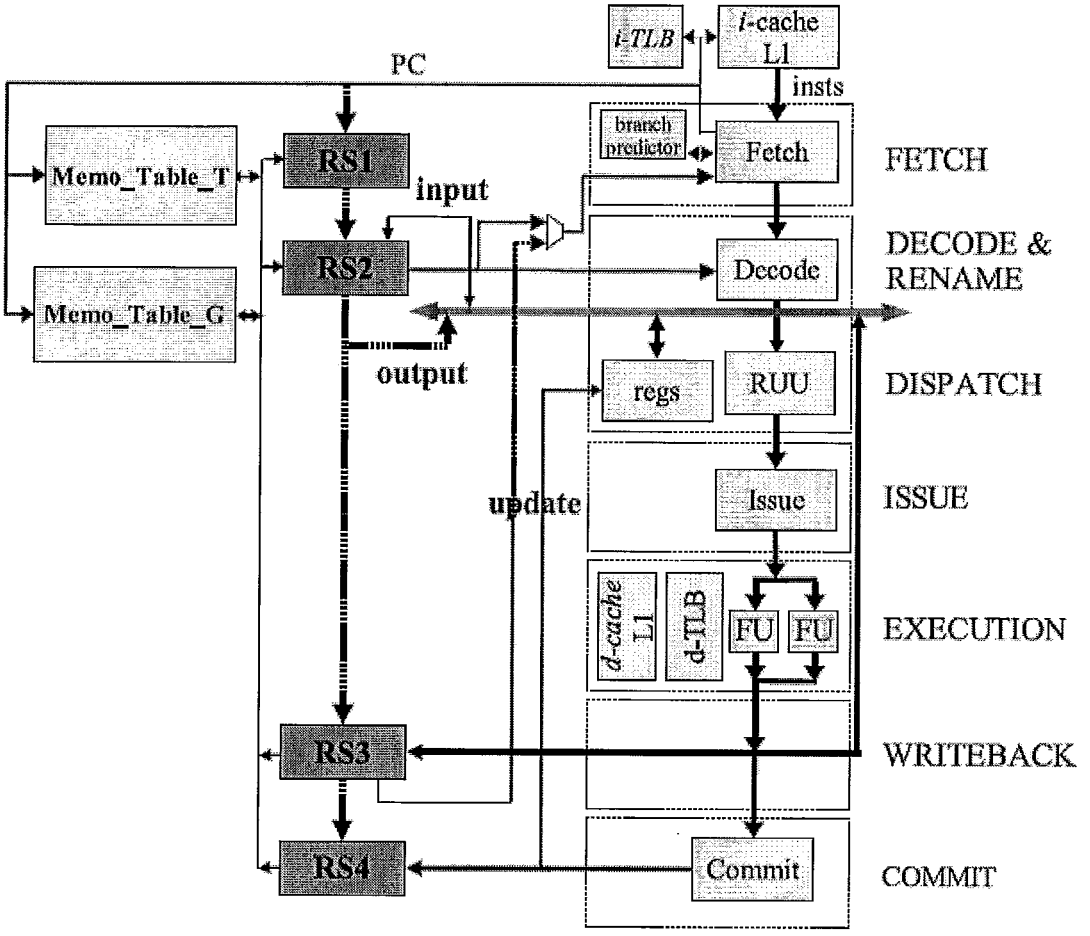


Figura 3. 6 – Pipeline RST

- **Os estágios do pipeline RST**

Como mencionado anteriormente, o estágio RS1 trabalha de forma semelhante ao estágio DS1 do DTM. Este estágio é responsável pela busca dos candidatos ao reuso nas tabelas de memorização. O PC da instrução corrente é enviado para o RS1, para que as tabelas sejam acessadas. Neste estágio do RST, instruções reusáveis e traços candidatos são recebidos do estágio RS4. Estes registros são gravados quando nenhuma entrada é encontrada nas tabelas de memorização.

O segundo estágio do pipeline RST difere de DS2, uma vez que o RST permite que traços que não possuem todas as entradas prontas possam ser reusados especulativamente, utilizando valores previamente armazenados em um traço. Para o reuso especulativo, o RST usa um mecanismo de confiança, que é utilizado para

verificar se o valor deve ser predito ou não. O objetivo do mecanismo de confiança é diminuir a quantidade de predições incorretas e suas penalidades associadas.

Para que o mecanismo de confiança fosse implementado, modificações foram feitas nos estágios RS1 e RS2. As tabelas de confiança podem ser acessadas juntamente com o acesso aos traços (em RS1) ou o acesso pode ser deixado para o momento onde há uma verdadeira necessidade de se usar o mecanismo de confiança (em RS2). Esta alternativa vai depender da profundidade do pipeline considerado, uma vez que o número de ciclos para acessar o mecanismo de confiança e atualizá-lo pode ser um ponto a ser analisado na escolha.

No estágio RS2, os registradores que serão acessados para o reuso de um traço estão incluídos como registros da Memo_Table_T, mas para as instruções são usados os valores decodificados no pipeline de instruções. Os valores dos registradores são comparados com instâncias da Memo_Table_T e Memo_Table_G e um traço só é predito se o mecanismo de confiança permitir.

Se a arquitetura utilizada não permitir a renomeação de registradores para resolver as dependências de dados, cada valor do contexto de saída, independente de ter seu reuso especulativo ou não, ocupará um entrada no Reorder Buffer e outra entrada em uma estação de reserva. As próximas instruções não precisam acessar os valores intermediários que foram criados dentro de um traço, apenas o valor final de cada entrada no contexto de saída do traço.

O estágio RS3 manipula o teste de predições incorretas. Neste estágio os resultados de instruções preditas são buscadas no Estágio de Escrita e seus valores preditos são comparados com os valores reais. Se uma predição incorreta for encontrada, a busca é redirecionada para o início do traço e todas as instruções posteriores ao traço são descartadas.

Além de armazenar novas predições e informações sobre recuperação, o estágio RS3 também é responsável por testar os valores provindos da fase de escrita para testar as predições incorretas. As saídas de RS3 são o modo especulativo para aquele traço, que é enviado para o estágio de entrega e um possível novo PC, que é enviado para o estágio de busca se uma predição incorreta é encontrada.

Para usar a predição em mais de uma entrada do traço, vários testes são realizados para um mesmo traço e uma tabela é utilizada para manipular tais testes. Esta tabela é chamada de Tabela de Recuperação (RT). Um contador é adicionado às entradas da tabela e é decrementado a cada vez que uma predição correta é detectada.

No caso de uma previsão incorreta, a tabela deve ser acessada para descartar todas as entradas que estão relacionadas aos traços preditos após um traço predito incorretamente. A informação sobre as predições são gravadas em ordem e as entradas da RT possuem os campos abaixo indicados. Uma entrada de RT é apresentada na Figura 3.7 (PILLA,2004).

Os campos da Tabela de Recuperação do RST são:

- *prid* é a identificação do produtor daquele valor
- *crid* é a identificação do consumidor daquele valor
- *val* é o valor predito, que será comparado com o valor calculado
- *lru* é um ponteiro para a entrada do traço em Memo_Table_T, que será atualizado em caso de uma previsão correta
- *cid* é um ponteiro para a entrada na tabela de confiança
- *np* é o número de predições feitas para o traço atual menos um
- *val* é um bit que define se a entrada é válida ou não

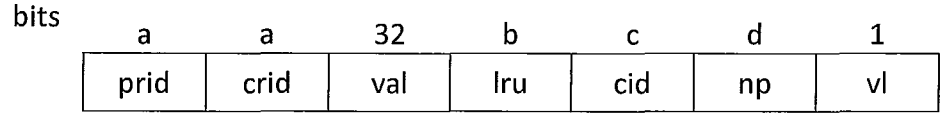


Figura 3. 7 – Exemplo de um registro da Tabela de Recuperação(RT)

A Figura 3.8 (PILLA,2004) ilustra a integração de todos os estágios do RST. O último estágio do RST é equivalente ao DS3 do DTM. No RS4 são detectados e armazenados instruções e traços redundantes. Dependendo da política utilizada, apenas instruções reusadas podem formar traços ou instruções do domínio de instruções reusáveis que ainda não foram reusadas podem ser incluídas. Todas as instruções posteriores a um traço predito incorretamente são descartadas neste estágio e a busca é redirecionada para o endereço inicial do traço.

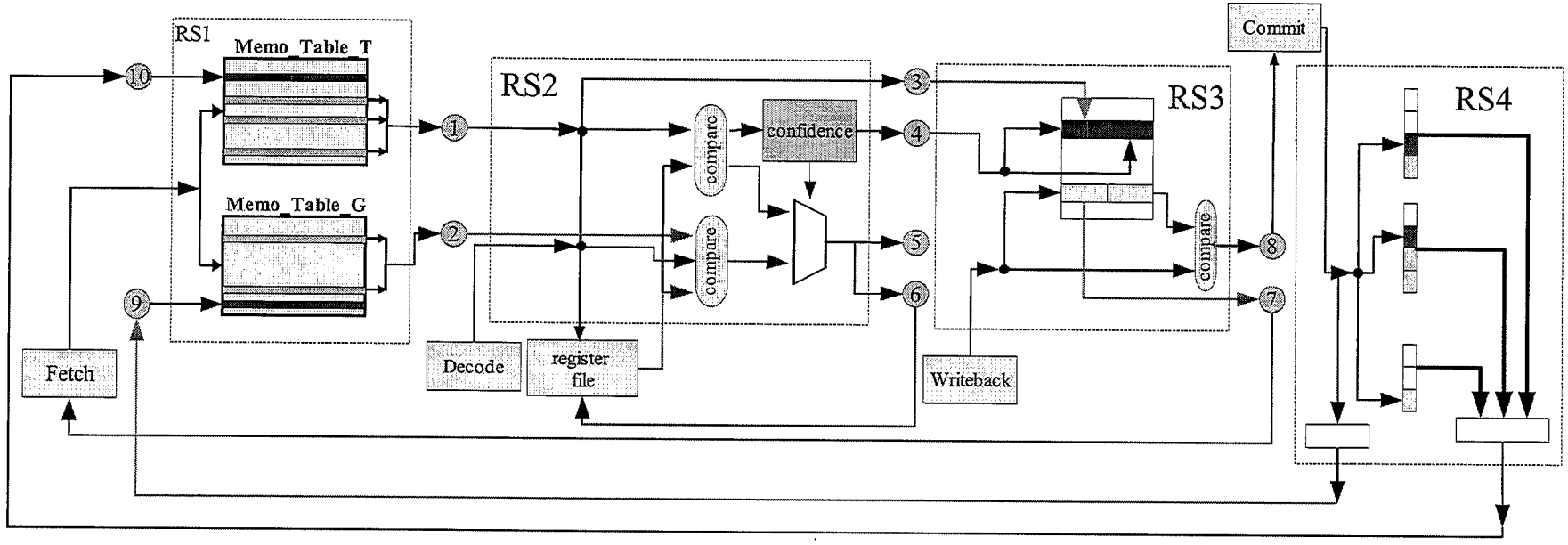


Figura 3. 8 – Integração entre os estágios do pipeline RST

4. Base Experimental, Resultados e Avaliações

Neste capítulo será apresentado o ambiente utilizado durante as simulações, que originou os resultados que foram analisados nesta etapa do trabalho. As características do ambiente de simulação, programas de teste utilizados e os parâmetros arquiteturais do simulador são apresentados na Seção 4.1. Na Seção 4.2, a metodologia experimental é explicitada. Em seguida, os resultados encontrados nos experimentos são apresentados na Seção 4.3. Por fim, a partir da Seção 4.4, as avaliações dos resultados são realizadas.

4.1. Base Experimental

4.1.1. Ambiente de Simulação

Para a execução das simulações, foram utilizadas estações Intel(R) Pentium(R) D CPU 3.00GHz, operando Linux Ubuntu 8.10.

O simulador utilizado nas execuções foi o sim-rst (PILLA,2004), uma versão alterada do simulador sim-outorder do SimpleScalar ToolSet (BURGER; AUSTIM, 1997; AUSTIM; LARSON; ERNST, 2002), versão 3.0b. Esta versão do simulador utilizou parte do código do simulador DTM original, que por sua vez, foi implementado utilizando a versão 2.0 do SimpleScalar ToolSet.

O sim-outorder original foi modificado para permitir o reuso de instruções e traços de instruções, sendo para isso incorporadas ao código do simulador original tabelas de memorização do mecanismo DTM, além dos estágios necessários para a implementação do mecanismo.

Algumas modificações foram feitas no mecanismo DTM, para a implementação do RST (PILLA; 2004). A fim de possibilitar especulação de valores, novos recursos foram incorporados ao simulador DTM, que além de permitir o reuso especulativo de traços, também torna possível o aumento do número de fases do pipeline para um número de estágios arbitrário, para processadores com pipelines mais longos que o pipeline de seis estágios original.

4.1.2. Benchmarks

Os programas de teste utilizados fazem parte do conjunto de benchmarks SPECInt2000, cuja finalidade é testar o desempenho de memórias e processadores. Os programas utilizados nas simulações são Bzip, Gcc, Gzip, Mcf, Parser e Vortex.

A Tabela 4.1 apresenta a relação dos benchmarks usados nas simulações, as entradas utilizadas para cada execução, bem como a média do número de instruções executadas entre as diferentes configurações de reuso, que serão explicitadas na subseção 4.2.1.

Tabela 4. 1 - Benchmarks usados nos experimentos

Benchmark	*Input*	*Média de instruções executadas*
Bzip	Input.Source	2.000.000.000
Gcc	200.i	2.500.000.000
Gzip	Input.Source	2.500.000.000
Mcf	inp.in	1.700.000.000
Parser	ref.in	2.500.000.000
Vortex	lendian1.raw	1.700.000.000

Todos os programas de teste foram compilados para a arquitetura PISA (Portable Instruction Set Architecture), um superconjunto do MIPS, utilizando o *cross-compiler* do gcc, versão 2.7.2.3 e glibc 1.09 do SimpleScalar 3.0 ToolSet, modo little-endian, para arquitetura x86. A arquitetura PISA foi eleita em decorrência do suporte no ToolSet do SimpleScalar.

Como informação complementar, a Tabela 4.2 apresenta a distribuição percentual dos tipos de instruções executadas para cada benchmark utilizado na simulação. As instruções estão divididas entre loads e stores, que são as instruções de acesso à memória, instruções de desvio, onde se incluem os desvios condicionais, incondicionais, diretos, relativos, chamada à subrotinas e retorno à subrotinas, instruções lógicas e aritméticas e outras instruções, que incluem as chamadas ao sistema operacional, nops e demais instruções.

Tabela 4. 2 – Distribuição das instruções executadas por tipo de instrução

Benchmark	*Acesso à Memória*	*Desvios*	*Lógicas e Aritméticas*	*Outras*
Bzip	32,92%	14,46%	51,33%	1,29%
Gcc	39,32%	20,09%	38,98%	1,62%
Gzip	30,44%	15,32%	50,97%	3,26%
Mcf	34,59%	28,16%	36,74%	0,51%
Parser	35,40%	23,12%	40,43%	1,05%
Vortex	56,24%	16,57%	26,84%	0,36%

4.1.3. Parâmetros Arquiteturais do Processador Simulado

Visando simular as características encontradas nos processadores superescalares atuais, modificações foram feitas nos parâmetros de configuração do simulador utilizado nas simulações. A Tabela 4.3 relaciona tais configurações.

Tabela 4. 3 – Configurações arquiteturais do processador utilizado nas simulações

Configuração	*Valores*
Largura do Pipeline	4 instruções
Cache de Instruções(L1)	64 bytes, 128 conjuntos, associatividade de 4, latência de 5 ciclos para acerto ao cache L1 e latência de 10 ciclos para acerto ao cache L2.
Cache de Dados(L1)	128 bytes, 64 conjuntos, associatividade de 4 ,latência de 5 ciclos para acerto ao cache L1 e latência de 10 ciclos para acerto ao cache L2.
Predição de Desvios	2 níveis, registradores de 13 bits(xor com PC), 8192 entradas
Unidades Funcionais	2 ULA de inteiros, 2 unidades de load e store, somadores de ponto flutuante, 1 unidade de multiplicação e divisão de inteiros, 1 ULA de ponto flutuante e 1 unidade de multiplicação e divisão de ponto flutuante.

4.1.4. Parâmetros Arquiteturais do Mecanismo DTM e RST

Do mesmo modo que configurações arquiteturais foram definidas para o processador utilizado nas simulações, alguns parâmetros de configuração foram definidos para os mecanismos DTM e RST, a fim de gerar os resultados das simulações de reuso nos benchmarks anteriormente mencionados. A Tabela 4.4 apresenta os parâmetros referentes aos mecanismos DTM e RST. Vale informar que a associatividade utilizada, assim como o tamanho do contexto de entrada e saída foram configurados de forma a não causar limitações nos resultados dos experimentos.

Tabela 4. 4 – Parâmetros dos mecanismos DTM e RST

Parâmetros	*Valores*	*Descrição*
dtm:allow_reusable	True	Permite instruções reusáveis no traço, ou seja, permite que se reuse instruções que não possuem entradas na Memo_Table_G, mas que podem ser instruções redundantes
dtm:notrace	False	Desabilita o reuso de traços
dtm:noinst	False	Desabilita o reuso de instruções
dtm:min_trace_size	2	Tamanho mínimo de instruções que compõem um traço a ser gravado
dtm:max_trace_size	64	Tamanho máximo de instruções que compõem um traço a ser gravado
dtm:inputscope	4	Operandos de entrada do traço
dtm:outputscope	4	Operandos de saída do traço
rst:memo_t_assoc	4	Associatividade da Memo_Table_T
rst:memo_t_sets	128	Número de conjuntos da Memo_Table_T
rst:memo_g_assoc	4	Associatividade da Memo_Table_G
rst:memo_g_sets	512	Número de conjuntos da Memo_Table_G
Memo_Table_G Size	32,7KB	Tamanho da entrada: 131 bits; Número de entradas: 2048
Memo_Table_T Size	25,2KB	Tamanho da entrada: 404 bits; Número de entradas: 512

Conforme é possível notar, o simulador foi configurado para permitir que um traço seja formado por no mínimo duas instruções. O reuso de apenas um subconjunto de instruções será bastante impactado por esta configuração, uma vez que há uma dificuldade muito grande de existirem duas ou mais instruções reusáveis do mesmo tipo em seqüência. Tal situação agrava-se mais ainda quando o subconjunto de reuso inclui somente instruções de acesso à memória, uma vez que este tipo de instrução caracteriza-se como delimitador da criação de traços, o que poderá ser comprovado nos resultados e análises comparativas, apresentados em subseções posteriores.

4.2. Metodologia Experimental

4.2.1. Motivação

A idéia destas simulações é verificar, dentro do domínio de instruções reusáveis do DTM, qual a contribuição de diferentes subconjuntos de instruções para o reuso. Os subconjuntos utilizados nas simulações foram divididos entre as instruções lógicas e aritméticas, instruções de desvio e cálculo de endereços das instruções load e store. As instruções aritméticas do tipo multiplicação não foram contempladas, devido ao custo extra de mais um registrador de saída. A relação do domínio de todas as instruções reusáveis é apresentada na Tabela 4.5.

Numa primeira etapa, os benchmarks foram executados utilizando apenas um subconjunto de instruções, ou seja, a partir do domínio de instruções reusáveis já pré-determinado, foi eleito apenas um grupo de instruções que teria a função de reuso ativada. As demais instruções do domínio não seriam reusadas neste momento. Em um segundo momento, a ativação para reuso destes subconjuntos foi feita de forma inversa, ou seja, simulações foram feitas reusando-se todas as instruções à exceção das instruções lógicas e aritméticas, todas as instruções do domínio menos os desvios e todas as instruções, excetuando-se o cálculo de endereços de load e store. Finalmente, uma tomada de execuções com o recurso do reuso ativado para todas as instruções do domínio foi feita, para fins comparativos.

Tabela 4. 5 – Domínio de instruções reusáveis

Instruções Registrador Registrador	*SubConjunto*	*Instruções Registrador Imediato*	*SubConjunto*
BEQ	Instrução de Desvio	JUMP	Instrução de Desvio
BNE	Instrução de Desvio	JAL	Instrução de Desvio
BLEZ	Instrução de Desvio	JR	Instrução de Desvio
LB_RR	Load Store	JALR	Instrução de Desvio
LBU_RR	Load Store	BGTZ	Instrução de Desvio
LH_RR	Load Store	BLTZ	Instrução de Desvio
LHU_RR	Load Store	BGEZ	Instrução de Desvio
LW_RR	Load Store	LB	Load Store
SB_RR	Load Store	LBU	Load Store
SH_RR	Load Store	LH	Load Store
SW_RR	Load Store	LHU	Load Store
ADD	Aritmética de inteiros	LW	Load Store
ADDU	Aritmética de inteiros	SB	Load Store
SUB	Aritmética de inteiros	SH	Load Store
SUBU	Aritmética de inteiros	SW	Load Store
AND	Lógica	ADDI	Aritmética de inteiros
OR	Lógica	ADDIU	Aritmética de inteiros
XOR	Lógica	ANDI	Lógica
NOR	Lógica	ORI	Lógica
SLLV	Lógica	XORI	Lógica
SRLV	Lógica	SLL	Lógica
SRAV	Lógica	SRL	Lógica
SLT	Lógica	SRA	Lógica
SLTU	Lógica	SLTI	Lógica
		SLTIU	Lógica
		LUI	Outras

4.3. Resultados

4.3.1. Medidas

Foram utilizadas as seguintes medidas para realizar a análise comparativa dos resultados das execuções dos benchmarks:

i. **Aceleração (speedup):** medida de desempenho que foi calculada a partir do total de ciclos sem reuso sobre o total de ciclos com a configuração de reuso em referência.

ii. **Taxa de Reuso:** valor do resultado dtm_reusable_rate. Este parâmetro corresponde ao total de instruções reusadas sobre todas as instruções que foram entregues. Esta medida inclui o reuso tanto de traces quanto de instruções isoladas.

iii. **Índice de Eficiência (IE):** Reusar todas as instruções pertencentes ao domínio de instruções reusáveis permite obter ganhos em aceleração, como já foi comprovado em trabalhos anteriores como o DTM (COSTA,2001) e o RST (PILLA,2004). Como o objetivo deste trabalho de dissertação é analisar quão sensível o mecanismo do reuso de traços pode ser aos subconjuntos de instruções, seria necessário questionar e comparar: se o ganho obtido em aceleração com a configuração de Reuso Total fosse analisado em relação ao percentual de instruções que foram executadas e entregues, haveria alguma maneira de se obter acelerações equivalentes, porém reusando um subconjunto menor de instruções? Se for possível obter uma aceleração similar à obtida na configuração de Reuso Total, considerando-se para reuso apenas um determinado subconjunto de reuso de instruções correspondente a apenas 15% do total de instruções executadas, por exemplo, este subconjunto não seria mais eficiente do que o conjunto de todas as instruções do domínio reusável? Partindo-se dessas questões, foi criado um Índice de Eficiência (IE), objetivando analisar o desempenho de cada subconjunto de instruções. Para totalizar o percentual de instruções para cada subconjunto de reuso analisado neste trabalho, o código do simulador foi alterado, com a introdução de um contador de instruções. Este contador totalizou o número de instruções reusadas e entregues, de cada subconjunto do domínio de instruções reusáveis que estava ativo para reuso no momento da simulação. A partir deste totalizador, foi possível conhecer o percentual da quantidade de instruções executadas

que este subconjunto representa com relação ao domínio completo de instruções reusadas. Para a análise da eficiência do reuso, este percentual foi relacionado à taxa de aceleração, ou seja, a aceleração obtida com o reuso de um determinado subconjunto de instruções foi analisada proporcionalmente ao percentual de instruções reusáveis com relação a todo o domínio de instruções reusáveis. Deste modo, o IE foi obtido da divisão da aceleração pelo percentual de instruções do subconjunto de reuso em questão:

$$IE = \frac{Aceleração\ do\ reuso\ do\ subconjunto}{percentual\ de\ instruções\ executadas\ no\ subconjunto}$$

iv. **Média Harmônica:** como uma melhor forma de comparar e analisar todas as formas de reuso de maneira mais normalizada, a média harmônica dos resultados também foi incluída nas análises comparativas. Quando se analisa o desempenho de computadores de uma forma geral, geralmente são apresentados conjuntos de valores por tarefa finalizada, como a análise de um conjunto de benchmarks, onde os tempos são gastos na execução de várias tarefas. Para taxas apresentadas por benchmark, a média harmônica é considerada a métrica mais apropriada (JACOB;MUDGE,1995). A Média Harmônica é dada pela seguinte expressão:

$$MH = n\left(\sum_{i=1}^{n}\left(\frac{1}{S_i}\right)\right)^{-1}$$

onde S_i indica qual é o elemento que está sendo considerado pela média.

4.3.2. Aceleração

Um dos parâmetros comparativos para esta análise foi a aceleração de desempenho. Para o cálculo desta medida, foi utilizado o resultado da divisão do número de ciclos para uma configuração sem reuso sobre a configuração do reuso do subconjunto em análise. Os gráficos das Figuras 4.1, 4.2 e 4.3 apresentam as medidas de aceleração comparando o subconjunto de instruções, seu inverso com relação ao domínio de instruções reusáveis e o Reuso Total, que engloba o reuso de todas as instruções. A aceleração foi medida para todos os benchmarks das simulações e a média harmônica destes, representada por MH nos gráficos, também é apresentada. As Tabelas

A.1, A.2 e A.3, do Anexo A, apresentam os valores de aceleração encontrados para todos os subconjuntos de reuso.

Para o primeiro subconjunto de instruções analisado, todos os benchmarks possuem um comportamento similar, com exceção de Gzip e Mcf, no que concerne à aceleração resultante do Reuso Total, se comparada ao reuso de todas as instruções menos Add e Sub. O gráfico da Figura 4.1 ilustra os resultados. O Reuso Total apresenta uma aceleração em média 2% maior do que no reuso de todas as instruções menos Add e Sub. Nos benchmarks Gzip e Mcf, a aceleração encontrada no reuso de todas as instruções menos Add e Sub é maior do que no Reuso Total e os valores encontrados são, respectivamente, 7,42% e 0,38% maiores. Na Média Harmônica, a aceleração do Reuso Total, quando comparada com o subconjunto de instruções lógicas e aritméticas Add e Sub é 7,84% maior e se comparada com o seu inverso, ou seja, o reuso de todas as instruções menos Add e Sub, onde é superior em 2,05%.

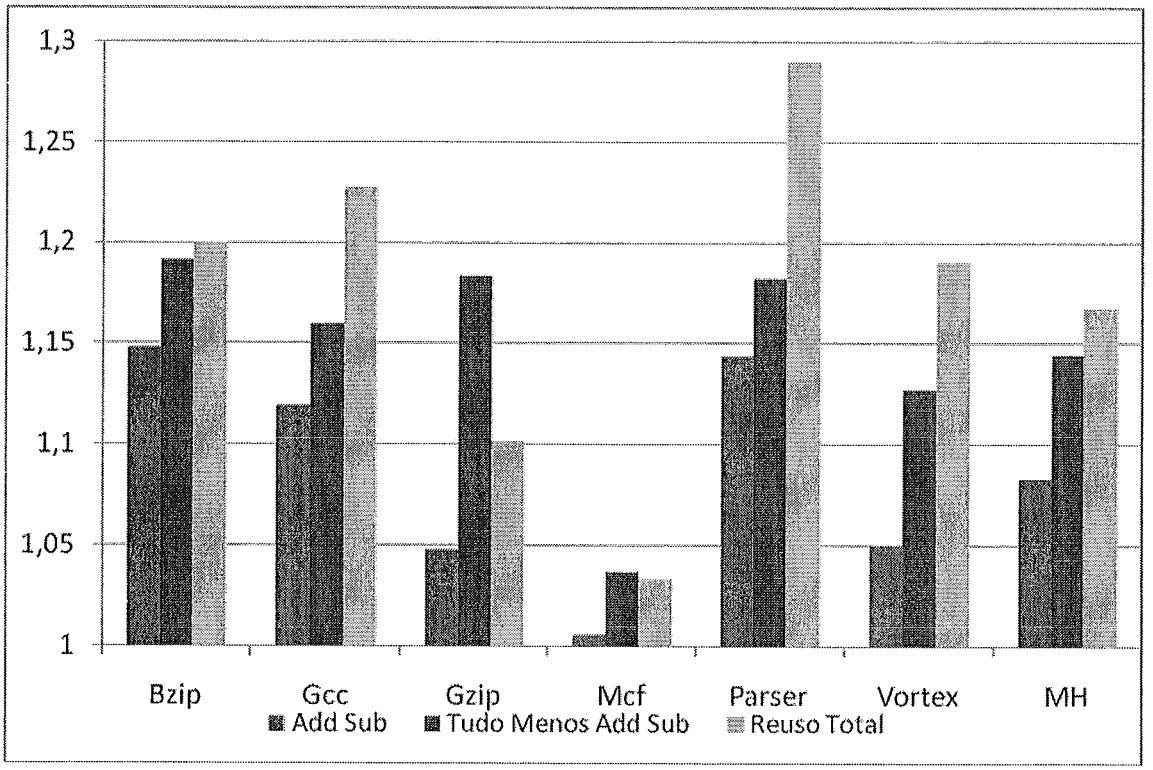


Figura 4. 1 – Aceleração Add Sub

Fazendo o comparativo do desempenho para o subconjunto de desvios (Figura 4.2), é possível observar que a aceleração na configuração do Reuso Total é em média

3% superior ao reuso de todas as instruções menos os desvios. Utilizando o subconjunto de reuso de desvios isoladamente como referência comparativa, a aceleração para o Reuso Total apresenta-se em média 11% superior ao subconjunto citado. A diferença mais expressiva é apresentada no Parser, onde o Reuso Total possui aceleração cerca de 20,8% superior ao subconjunto de desvios. A aceleração mostra-se sempre positiva no reuso de todas as instruções menos desvios, variando de 1,03 até 1,23 em Mcf e Parser, respectivamente. O Mcf, utilizando a configuração de reuso de desvios foi o único a apresentar uma desaceleração (0,9992). Apesar de ser um valor estatisticamente irrelevante, a razão desta perda de desempenho pode estar relacionada ao fato do simulador adicionar um ciclo a cada redirecionamento da busca que o reuso de um desvio provoque, acarretando um pequeno acréscimo no número final de ciclos da execução. Como esta medida possui um valor negativo quase insignificante, até mesmo o ambiente de simulação pode ter influência neste dado final. Tais informações são ilustradas no gráfico da Figura 4.2.

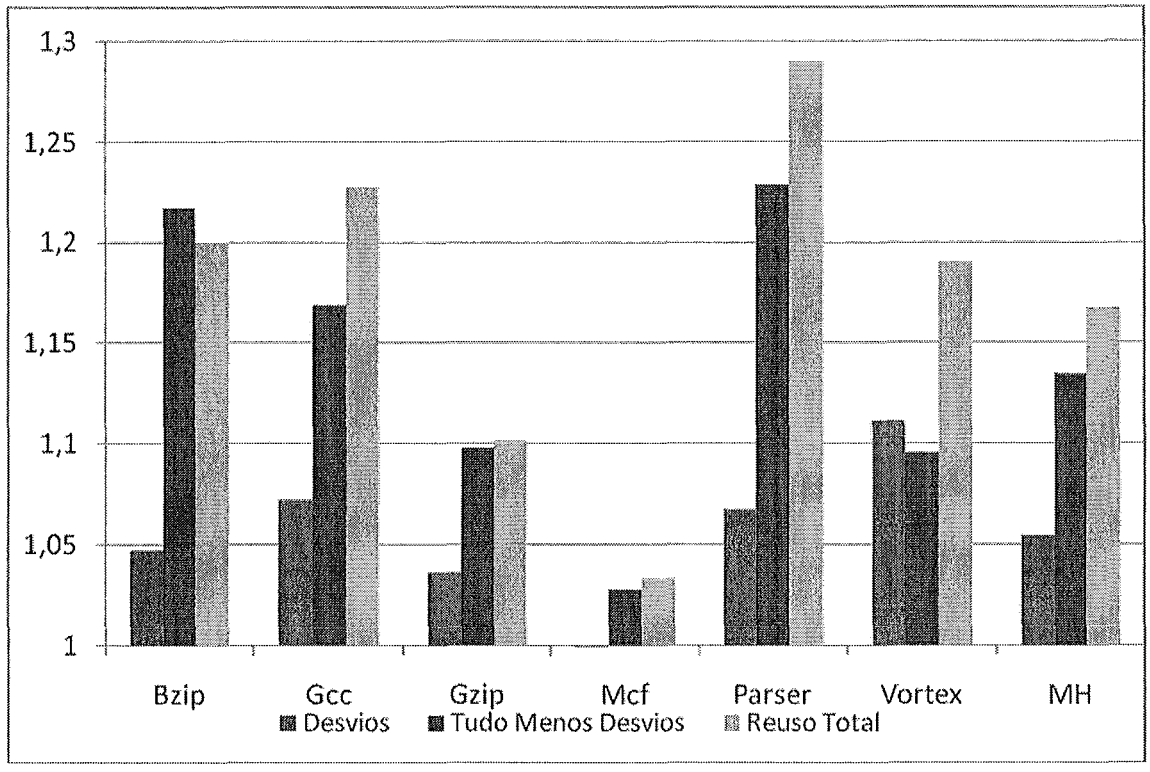


Figura 4. 2 – Aceleração Desvios

A Figura 4.3 apresenta o gráfico da análise do reuso do subconjunto Load e Store, onde as taxas de aceleração do Reuso Total apresentaram-se maiores que o reuso de todas as instruções menos Load e Store em uma média de 1,5%. Os benchmarks Gcc

e Vortex, apresentaram uma aceleração maior do que o Reuso Total para esta mesma configuração: 1,25% e 1,28%, respectivamente. A aceleração do Reuso Total, se comparada com o reuso das instruções Load e Store, exclusivamente, é em média 14,8% superior.

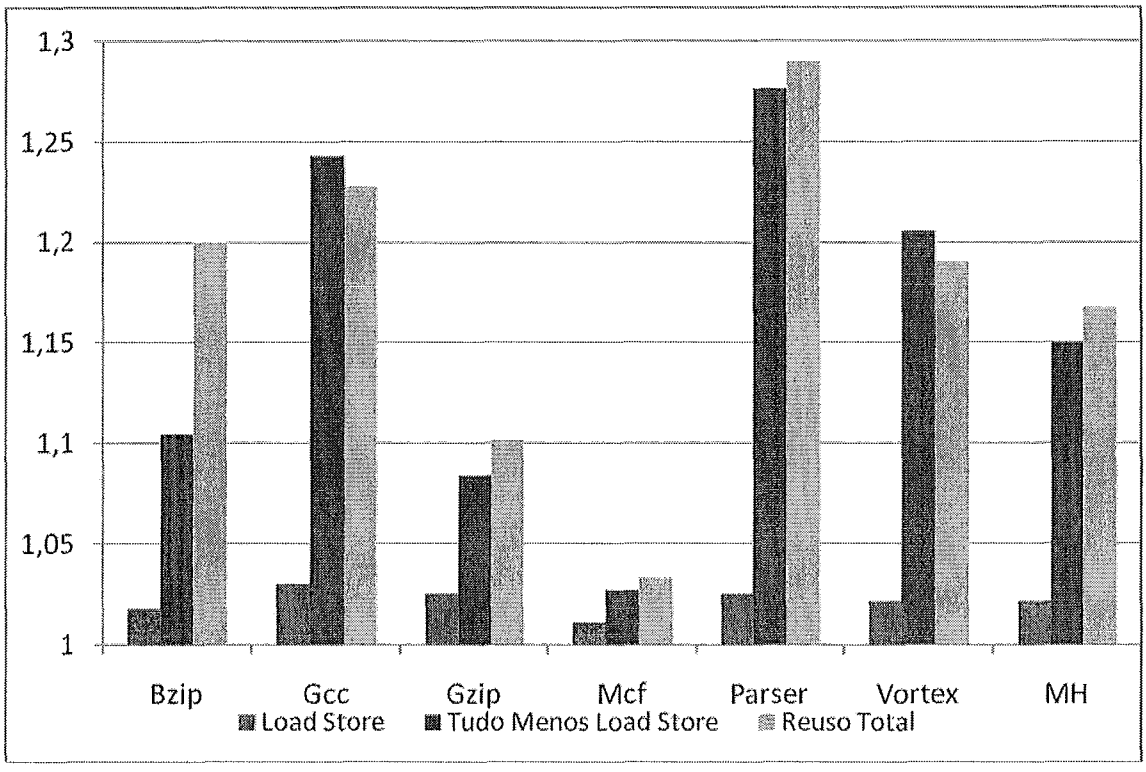


Figura 4. 3 – Aceleração Load Store

4.3.3. Taxa de Reuso

Outra medida utilizada nas avaliações foi a taxa de reuso, extraída dos logs de execução do simulador, no parâmetro ***dtm_reusable_rate***. Esta medida é resultante do total de instruções reusadas sobre o total de instruções entregues, como foi apresentado na seção anterior. A Média Harmônica também foi utilizada nesta etapa da análise. Os gráficos das Figuras 4.4, 4.5 e 4.6 apresentam, respectivamente, tais medidas para os subconjuntos de instruções Add e Sub, instruções de Desvios e Load e Store. Em todas as análises, de forma previsível, as taxas de reuso na configuração de Reuso Total apresentaram-se superiores às demais taxas de reuso dos subconjuntos de instruções avaliadas. As Tabelas A.4, A.5 e A.6, do Anexo A, explicitam as medidas de taxa de reuso encontradas nas simulações.

A Figura 4.4 ilustra o gráfico das taxas de reuso do subconjunto de instruções Add e Sub. A taxa que corresponde ao Reuso Total varia entre 0,26 e 0,41 e é em média superior em 235% às taxas de reuso de instruções Add e Sub, além de superar em 183% a taxa de reuso de todas as instruções menos Add e Sub. O subconjunto de instruções inverso, ou seja, Tudo Menos Add e Sub, possui taxa de reuso de em média 17,4%, contra 31,7% do Reuso Total.

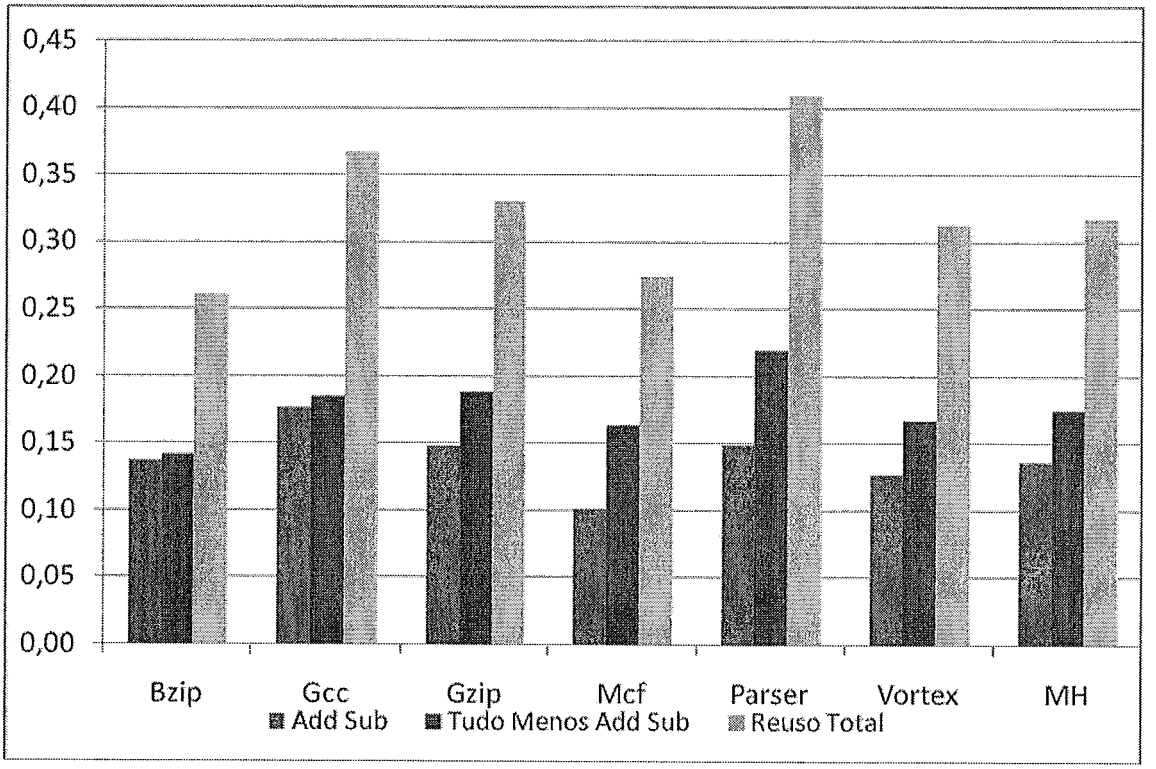


Figura 4. 4 – Add e Sub: Taxa de Reuso

Na Figura 4.5 são apresentadas as taxas de reuso para o subconjunto de desvios. Para todos os subconjuntos de reuso, o comportamento é similar. A taxa de reuso para a configuração de Reuso Total é sempre superior ao reuso de todas as instruções menos os desvios, que por sua vez, são superiores às taxas de reuso somente de desvios. As taxas de reuso de desvios variam de 0,027 a 0,085 no Bzip e no Parser, respectivamente. O Bzip é único benchmark a apresentar taxa de reuso de todas as instruções menos desvios (0,281) superior à taxa de reuso total (0,261). A configuração de Reuso Total apresenta taxas de reuso 83% superiores ao reuso exclusivo de instruções de desvio. Realizando a mesma comparação do Reuso Total agora com o reuso de todas as

instruções menos as instruções de desvio, esta diferença percentual se reduz consideravelmente (15%).

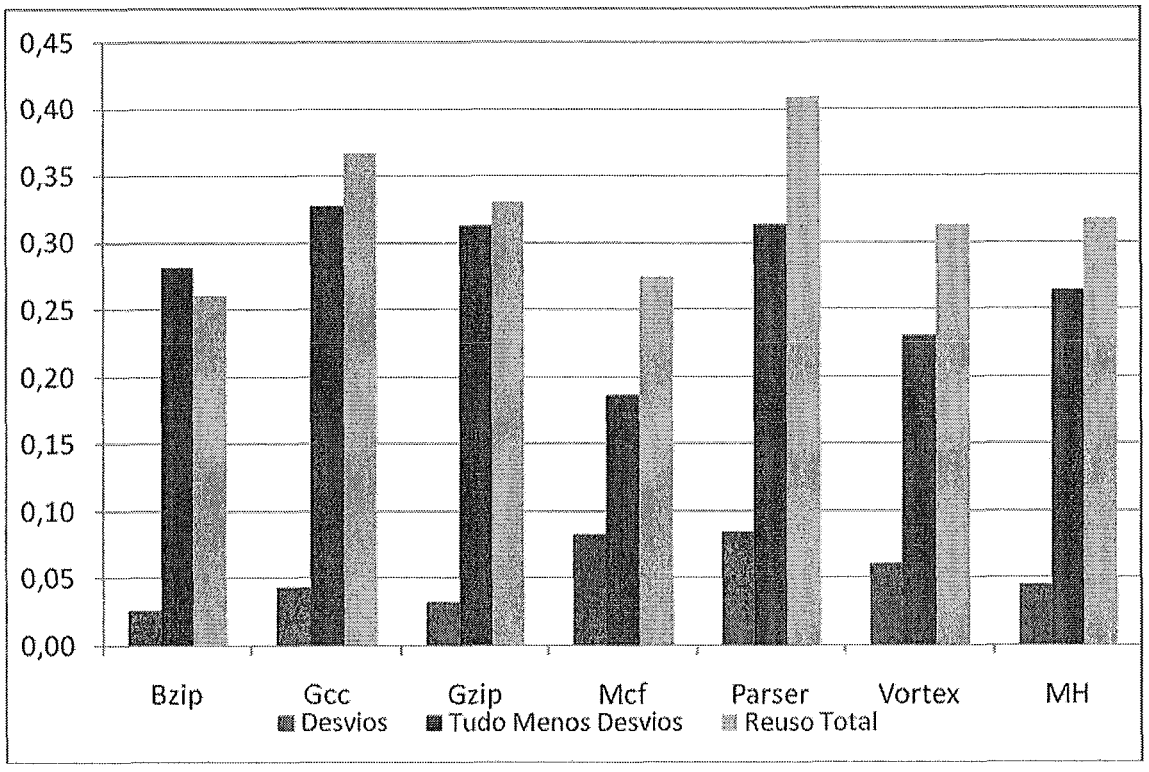


Figura 4. 5 – Desvios: Taxa de Reuso

A Figura 4.6 ilustra o gráfico das taxas de reuso do subconjunto de instruções de Load e Store. As taxas de reuso para as instruções Load e Store variam entre 0,05 e 0,11. Já as taxas de Reuso Total variam entre 0,26 e 0,41. O reuso de todas as instruções menos Load e Store obteve sua melhor taxa de reuso no benchmark Parser, atingindo a taxa de 0,29.

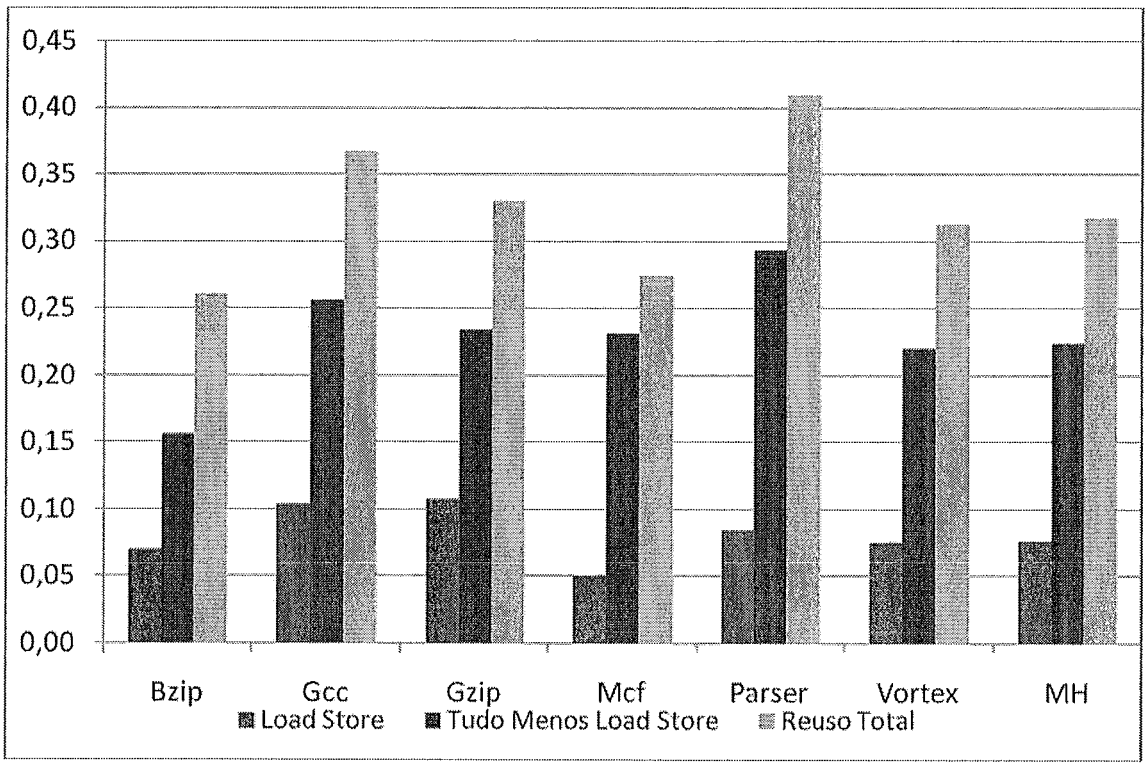


Figura 4. 6 – Load e Store: Taxa de Reuso

4.3.4. Índice de Eficiência

Sabendo-se que cada subconjunto de instruções possui uma particularidade, no que concerne à sensibilidade com relação ao mecanismo do reuso de traços, o índice de eficiência é a medida que permite uma análise mais conclusiva acerca dos benefícios de se eleger um subconjunto de instruções para reuso, ao invés de se trabalhar com o domínio de instruções reusáveis em sua íntegra. O IE possibilita a percepção do quanto cada subconjunto de reuso contribui com a taxa de aceleração final obtida através deste mecanismo, no momento em que relaciona o percentual de instruções reusadas com a aceleração obtida neste subconjunto de reuso. Os índices encontrados nesta análise são revelados nas Tabelas A.7, A.8 e A.9, do Anexo A.

Na Figura 4.7, os índices de eficiência obtidos a partir das análises do subconjunto de instruções lógicas e aritméticas, apresentados como Add e Sub, são ilustrados. É notório observar que em todos os casos, o Reuso Total mostra-se menos eficiente que o reuso de um subconjunto, seja ele apenas composto de instruções aritméticas ou o seu inverso. Reusar somente instruções Add e Sub apresenta um IE inferior ao seu subconjunto inverso em Bzip e Gzip, apresentando uma eficiência menor

em 9,44% e 17,44%, respectivamente. Nas demais situações, o reuso do subconjunto Add e Sub mostra-se em média 62% mais eficiente que o seu conjunto inverso e 121,4 % superior ao Reuso Total. Cabe informar que este índice de eficiência não é absoluto, tendo por fim apenas comparar as diversas configurações de reuso para os mesmos benchmarks.

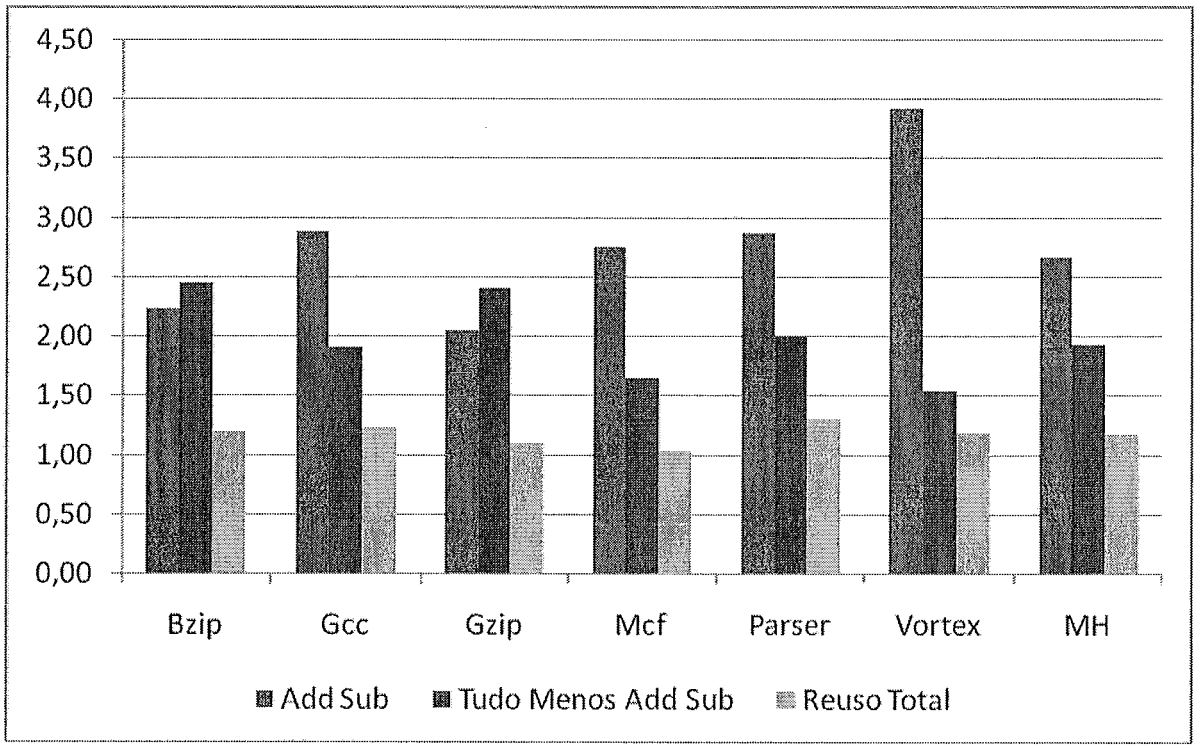


Figura 4. 7 – Índice de Eficiência Add e Sub

O subconjunto de reuso de Desvios foi o que obteve os maiores índices de eficiência atingindo superioridade com relação ao Reuso Total em todos os casos, com índices variando entre 3,58 e 7,26, enquanto os índices do Reuso Total variaram de 1,04 a 1,31. Isto se deve ao fato das instruções de desvios representarem cerca de 19,62% das todas as instruções executadas e uma aceleração média de 90,2% da aceleração apresentada no Reuso Total. Com taxas de reuso baixas e sem grandes perdas de aceleração o subconjunto de desvios apresenta ótimos índices de eficiência. O gráfico da Figura 4.8 mostra que o Bzip atinge um índice de eficiência de 7,26, enquanto seu conjunto inverso apresenta 1,43 e o Reuso Total 1,20.

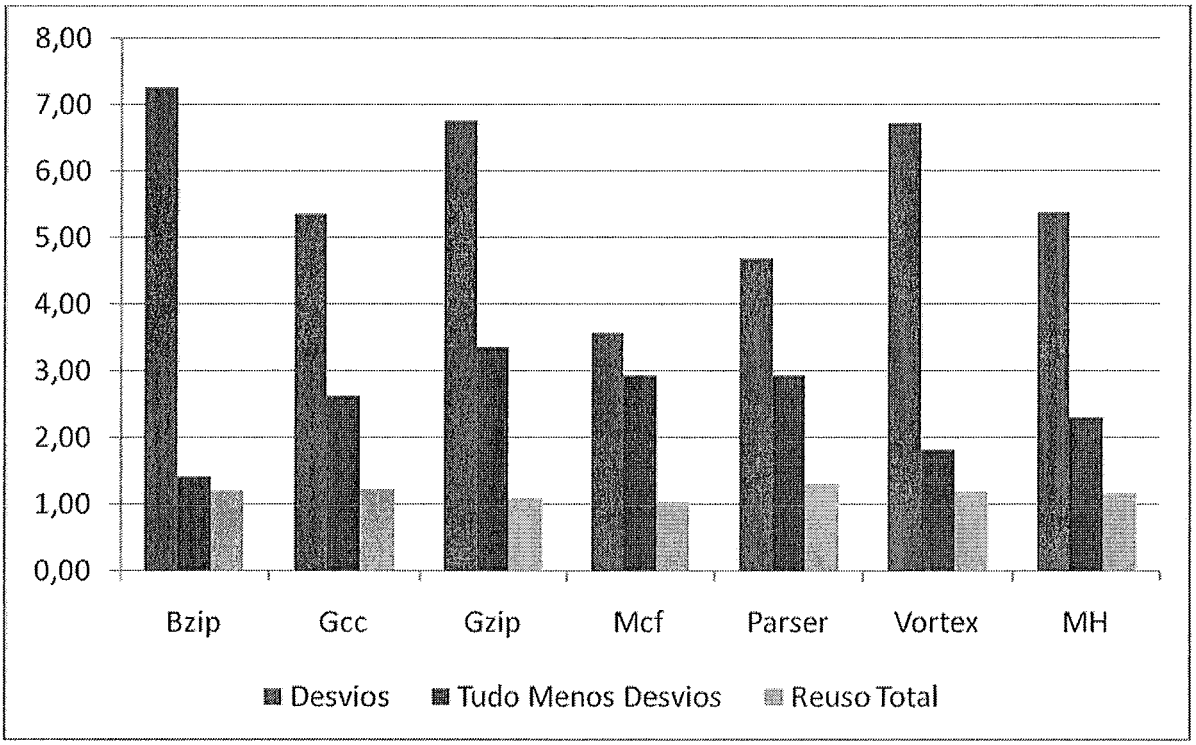


Figura 4. 8 – Índice de Eficiência Desvios

Por fim, o grupo de instruções Load e Store foi analisado com relação à sua eficiência no reuso. É possível observar, a partir do gráfico da Figura 4.9, que os índices médios de eficiência encontrados são próximos aos índices encontrados no subconjunto de instruções lógicas e aritméticas. Isto se deve ao fato de que os percentuais de instruções executadas em cada subconjunto são quase similares com relação ao total de instruções executadas. Dentro do total das instruções reusadas, o subconjunto Add e Sub representa em média 40,88%, contra 38,15% do subconjunto Load e Store. Uma vez que o índice de eficiência está diretamente relacionado a este percentual, é intuitivo que em média estes índices sejam similares.

Confrontando os índices de eficiência de Add e Sub com o subconjunto Load Store, é notável que os benchmarks que obtiveram um desempenho mais baixo com o reuso de Add e Sub, apresentam os melhores índices de eficiência no reuso de Load e Store. O Bzip e o Gzip apresentam os maiores índices: 3,10 e 3,37, respectivamente. O inverso também ocorre com o benchmark Vortex, que obteve o índice mais baixo para reuso de Load e Store (1,82), enquanto obteve seu maior índice de eficiência no reuso de Add e Sub, com 3,93.

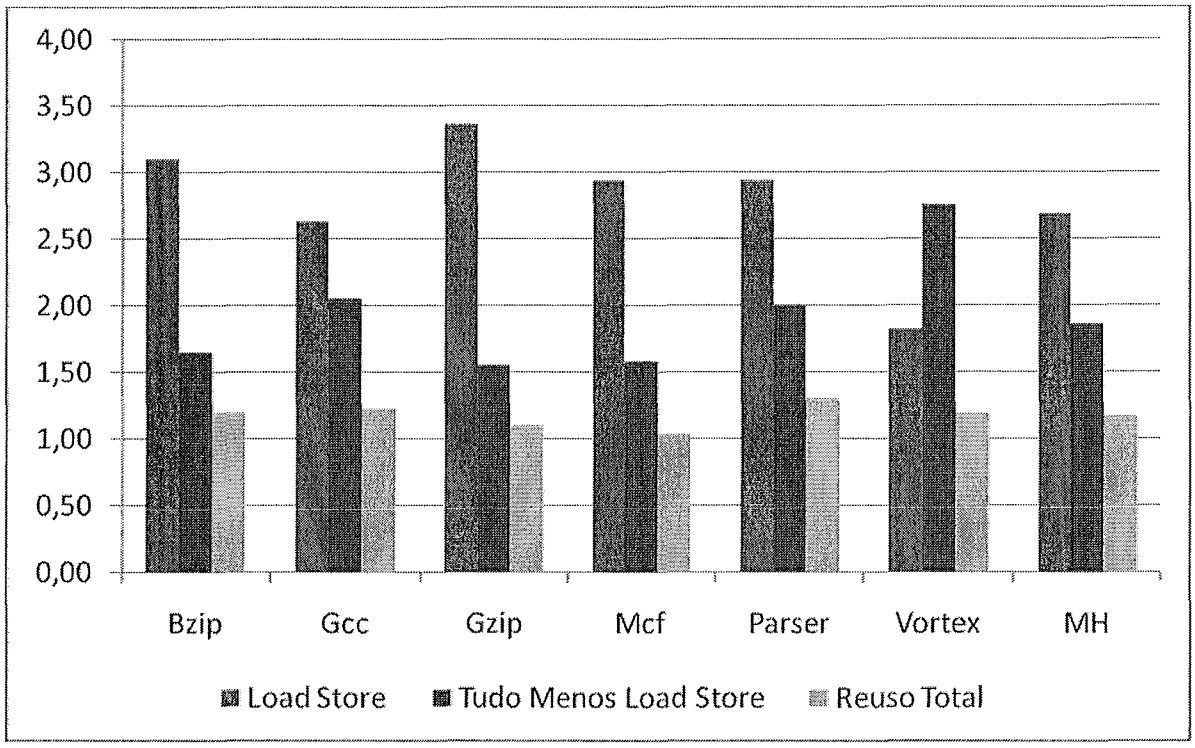


Figura 4. 9 – Índice de Eficiência Load e Store

A partir das medidas de aceleração, taxas de reuso e índices de eficiência apresentados na seção anterior, uma análise dos diversos subconjuntos de reuso pode ser realizada, objetivando comparar e, possivelmente destacar, um subconjunto de instruções reusáveis como mais eficiente, no que concerne ao desempenho. Ao perceber que subconjunto de reuso é mais sensível ao mecanismo do reuso de traços, é possível elencar alguns dados que possam ser relevantes numa escolha de um mecanismo de reuso que possa utilizar um domínio menor de instruções reusáveis sem grandes perdas de desempenho. Além disso, quando um número menor de instruções é reusado, uma taxa de reuso menor é encontrada, o que acarreta diretamente uma quantidade menor de acessos às tabelas de reuso e, como conseqüência, a redução do consumo de energia.

Em uma primeira análise, para o subconjunto de reuso Add e Sub, é possível observar que de todos os subgrupos, este é o que possui a melhor taxa de aceleração, como ilustrado na Figura 4.10. A taxa de aceleração para o reuso somente de instruções lógicas e aritméticas representa 92,8% da aceleração com Reuso Total. Este subconjunto de reuso consegue obter esta taxa de aceleração, apresentando uma taxa de reuso 42,8% menor do que no Reuso Total (Figura 4.13). Este subconjunto apresenta-

se como uma alternativa interessante de redução do domínio de reuso para traços, uma vez que atinge quase a totalidade da aceleração do Reuso Total, para taxas de reuso inferiores a 50%. A quantidade de acessos à Memo_Table_T neste subconjunto de instruções chega a 50,66% do total de acessos que seriam feitos no Reuso Total. O subconjunto de instruções Todas Menos Add e Sub, consegue atingir 97,9% da aceleração do Reuso Total, com uma taxa de 54,8% de reuso, o que garante uma eficiência igualmente interessante, comparando-se com o Reuso Total.

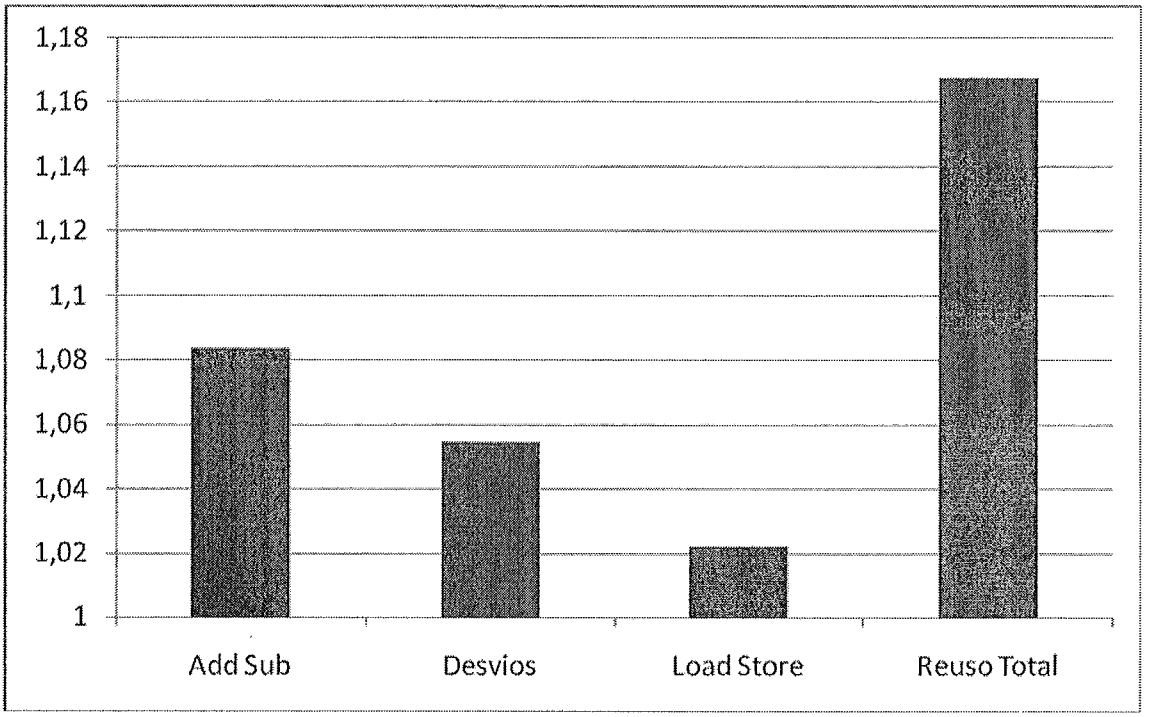


Figura 4. 10 – Média Harmônica da Aceleração

Algumas observações acerca do reuso do subconjunto de Desvios podem ser feitas. A partir dos dados apresentados nos índices de eficiência, apresentados no gráfico da Figura 4.14, nota-se que reusar desvios, apesar de não representar um percentual das instruções executadas muito significativo (19,6% de todas as instruções reusadas são instruções de desvios) em se comparando com os outros tipos de instruções, a aceleração obtida com este pequeno percentual representa cerca de 90,3% da aceleração do Reuso Total. Estes dados permitem concluir o motivo deste subconjunto de instruções possuir índices de eficiência tão satisfatórios, como ilustra a Figura 4.14. Tal comportamento leva a perceber que as instruções de desvios têm uma

importância muito grande dentro no contexto do reuso e que reusar somente instruções de desvios já representa um ganho considerável, se usarmos o índice de eficiência como referencial comparativo.

Ainda dentro do subconjunto de desvios, uma análise interessante sobre reusar o seu subconjunto inverso, ou seja, reusar todas as instruções menos as instruções de desvios pode ser feita. Se o domínio das instruções reusáveis excluísse todas as instruções de desvio, uma aceleração média de 97,2% do Reuso Total seria obtida, para uma taxa de reuso de, em média, 83,2% do total. A total de acessos à Memo_Table_T para o reuso exclusivo de instruções de desvio é 77,7% inferior à configuração de Reuso Total.

A estrutura das tabelas de reuso Memo_Table_G e Memo_Table_T inclui campos que são específicos para o reuso de desvios. Se estas instruções não forem elencadas dentro do novo domínio de reuso, estas tabelas terão seu tamanho físico reduzido com a exclusão destes campos. Deste modo, a redução do espaço de alocação de memória necessária para as tabelas de reuso seria outro ponto positivo dentro do subconjunto Tudo Menos Desvios.

Para calcular o quanto de redução poderia ser obtido, eliminando os campos referentes a desvios nas Memo_Table_G e Memo_Table_T, é necessário usar como referência tais estruturas de reuso nas dimensões utilizadas nos experimentos. Para a Memo_Table_G, o tamanho da entrada independe das configurações, necessitando apenas da exclusão dos campos jmp, brc e btaken. A estrutura da nova entrada para Memo_Table_G é apresentada na Figura 4.11. Como mostra a Tabela 4.6, a nova Memo_Table_G, sem o suporte aos reuso de desvios sofre uma redução de 2,1% em capacidade com relação à Memo_Table_G original.

Tabela 4. 6 – Memo_Table_G original e sem reuso de desvios

Memo_Table_G	Tamanho da entrada	Memo_Table_G sem desvios	Tamanho da entrada sem desvios
Número de conjuntos: 512 Tamanho da entrada: 131 bits Número de entradas: 2048 Associatividade: 4	pc = 32 bits sv1 = 32 bits sv2 = 32 bits res/targ = 32 bits jmp = 1 bit brc = 1 bit btaken = 1 bit	Número de conjuntos: 512 Tamanho da entrada: 128 bits Número de entradas: 2048 Associatividade: 4	pc = 32 bits sv1 = 32 bits sv2 = 32 bits res/targ = 32 bits
Tamanho Total: 32,7KB	131 bits	Tamanho Total: 32KB	128 bits

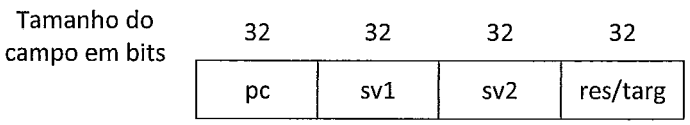


Figura 4. 11 – Entrada da Memo_Table_G sem reuso de desvios

Para a Memo_Table_T, o cálculo do tamanho de cada entrada, e como conseqüência, o tamanho total da tabela, depende do tamanho dos contextos de entrada e saída definidos no simulador, bem como o total de desvios por traço. O tamanho do contexto de entrada e saída dos traços foi previamente definido com 4, objetivando que este parâmetro não representasse uma limitação durante os experimentos. Pelo mesmo motivo, o valor máximo de desvios permitido por traços foi definido como ilimitado, permitindo qualquer número de desvios encontrado nos traços. Para todas as simulações realizadas, o maior valor de desvios encontrado para um traço foi igual a 22. Deste modo, para o cálculo do tamanho físico da Memo_Table_T, apresentados na Tabela 4.6, os valores de entrada e saída considerados foram iguais a 4 e o valor de desvios igual a 22. A Figura 4.12 exibe a proposta de uma nova estrutura da Memo_Table_T, sem suporte ao reuso de desvios. Como é possível perceber, tal modificação representa uma redução em alocação de memória para acesso à Memo_Table_T de 10,7%.

Tabela 4. 7 – Memo_Table_T original e sem reuso de desvios

Memo_Table_T	Tamanho da entrada	Memo_Table_T sem desvios	Tamanho da entrada sem desvios
Número de conjuntos: 128 Tamanho da entrada: 404 bits Número de entradas: 512 Associatividade: 4	pc = 32 bits npc = 32 bits icr = 5 x 4(input) icv = 32 x 4(input) ocr = 5 x 4(output) ocv = 32 x 4(output) bmask = 22(máximo de desvios) btaken = 22(máximo de desvios)	Número de conjuntos: 128 Tamanho da entrada: 360 bits Número de entradas: 512 Associatividade: 4	pc = 32 bits npc = 32 bits icr = 5 x 4(input) icv = 32 x 4(input) ocr = 5 x 4(output) ocv = 32 x 4(output)
Tamanho Total: 25,2KB	404 bits	Tamanho Total: 22,5KB	360 bits

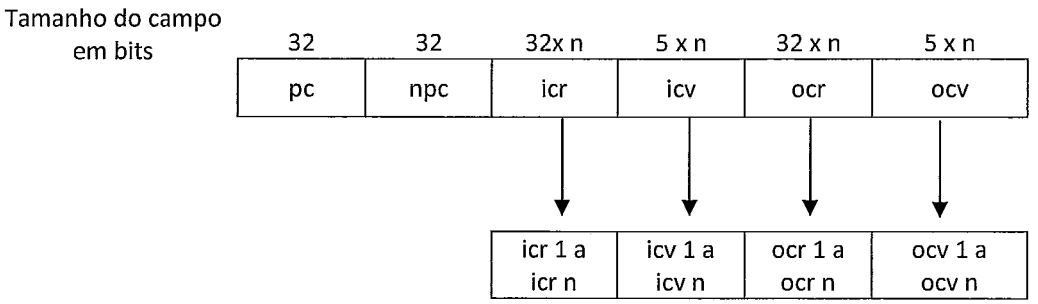


Figura 4. 12 – Entrada da Memo_Table_T sem reuso de desvios

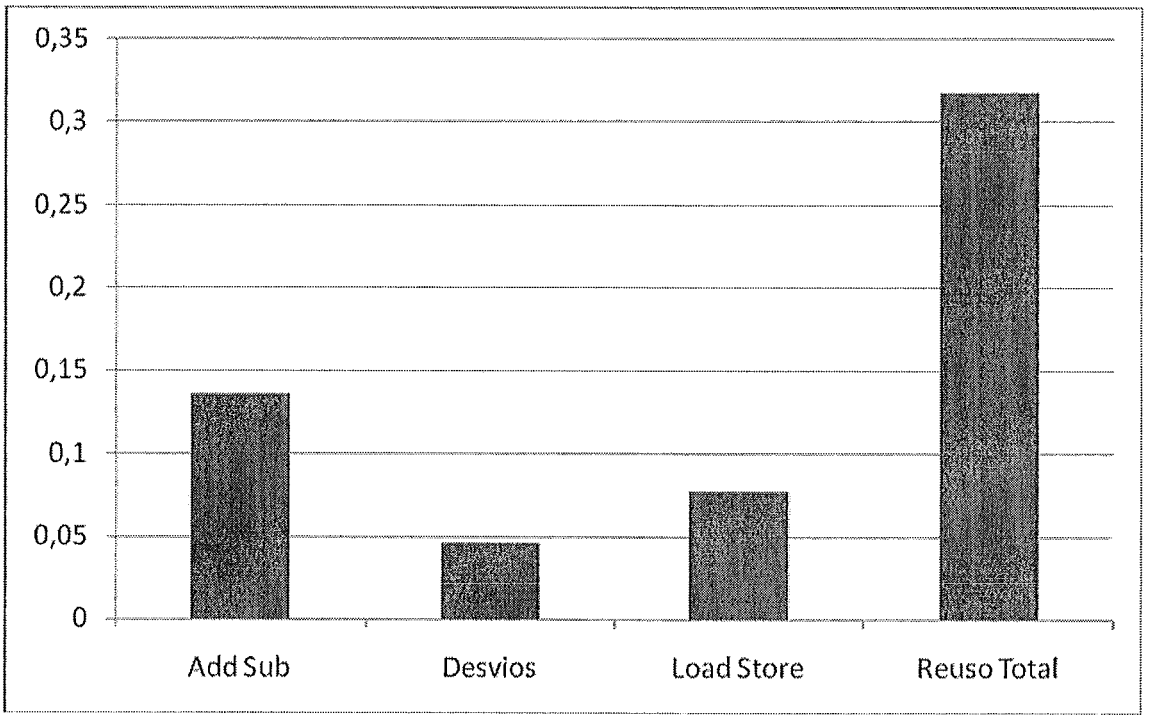


Figura 4. 13 - Média Harmônica das Taxas de Reuso

O último subconjunto de reuso a ser analisado é o subconjunto de instruções de acesso à memória Load e Store. Vale lembrar que o reuso destas instruções refere-se apenas a reusar cálculos de endereços. Se a escolha por um subconjunto de reuso objetivar um melhor desempenho no reuso de traços, este subconjunto isoladamente não deve ser o eleito. O modo como o mecanismo de reuso foi implementado no simulador define que um cálculo de endereços de Load ou Store determina o fim de um traço.

Desta maneira, não há possibilidade de que duas instruções em seqüência neste subconjunto formem um traço de tamanho mínimo de duas instruções, valor que foi configurado como parâmetro de tamanho mínimo dos traços nas simulações. Em decorrência de tal fato, o subconjunto Load e Store faz apenas o reuso de instruções isoladas.

Como a taxa de reuso representa todo o reuso obtido tanto em instruções isoladas como em traços, o subconjunto Load e Store, possui uma taxa de reuso média de 24,3% da configuração do Reuso Total, com uma aceleração de, em média, 87,5% do total, o valor mais baixo de todos os analisados, apresentando um IE apenas 0,75% superior ao do reuso de Add e Sub. Conforme já mencionado, o subconjunto Load e Store não faz acessos à Memo_Table_T, por não fazer reuso de traços. A solução para este subconjunto de reuso seria utilizar a versão do RST que utiliza uma tabela unificada de reuso (PILLA,2004), onde cada entrada da tabela é capaz de memoizar traços de uma instrução, ou seja, permite a memoização de instruções isoladas ou traços de mais de uma instrução. Isto representaria uma economia de 43,5% em alocação de espaço em memória, correspondente aos 25,2KB que a Memo_Table_T atingiu em capacidade nestas simulações.

Outro ponto relevante que deve ser levado em consideração quando se pensa em reduzir o domínio de instruções reusáveis é a economia de energia. Segundo cálculos realizados utilizando a ferramenta CACTI 5.2, disponível gratuitamente no site da Hewllet Packard, o consumo de uma leitura à Memo_Table_G é de aproximadamente 58,4 mW. Para a Memo_Table_T, o custo de uma leitura é de 327, 7mW. O consumo também foi calculado na única situação deste trabalho onde a tabela de reuso de traços poderia sofrer redução em capacidade. Quando se opta por reusar todas as instruções menos os desvios, uma leitura a esta tabela Memo_Table_T reduzida é de 320,6 mW. Apesar de a tabela ter seu tamanho físico reduzido em 10,7%, o consumo de uma leitura a essa tabela redimensionada reduz-se apenas em 2%, o que mostra que na realidade, não é o tamanho da tabela que deve ser reduzido, mas sim, a quantidade de acessos feitos a estas estruturas de reuso, diminuindo desta forma o custo total das leituras em conseqüência da redução dos acessos. A redução dos acessos torna-se possível através da redução do domínio de reuso.

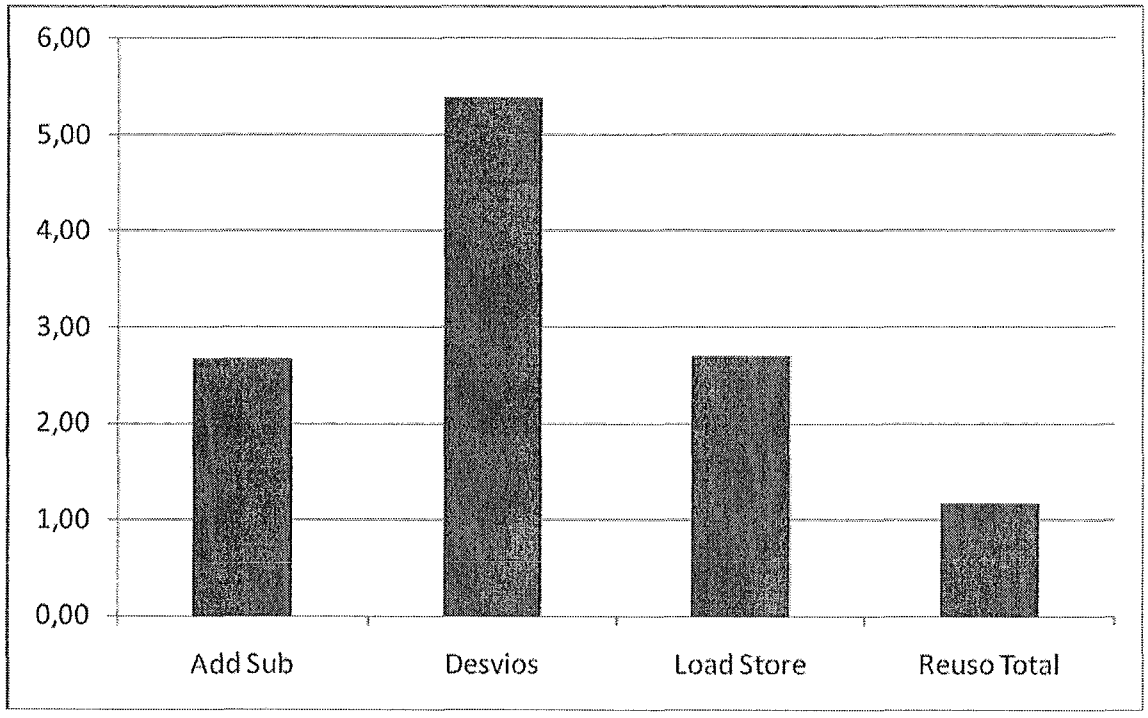


Figura 4. 14 – Média Harmônica dos Índices de Eficiência

5. Conclusões

Neste trabalho de dissertação objetivou-se analisar a sensibilidade que cada subconjunto de instruções tem ao mecanismo do reuso de traços. Os subconjuntos foram selecionados por tipo de instrução dentro do domínio de instruções reusáveis do DTM (COSTA,2001). Uma análise geral foi feita com os subconjuntos de instruções lógicas e aritméticas, instruções de desvio e instruções de acesso à memória, visando verificar os aspectos positivos e negativos que cada subconjunto possui dentro do contexto de desempenho em reuso de traços. A partir das simulações e análise das medidas encontradas, algumas conclusões podem ser feitas sobre os subconjuntos de reuso analisados comparativamente:

O subconjunto de reuso Add e Sub apresenta um desempenho satisfatório, no momento em que obtém quase que a totalidade da aceleração do Reuso Total(92,8%) com menos da metade da taxa de reuso (42,8%). Além disso, os acessos à Memo_Table_T atingem à faixa de 50,7% do total de acessos que seriam feitos no Reuso Total.

Para o subconjunto de Desvios, os índices de eficiência mostraram-se altamente satisfatórios. Isto se deve ao fato da taxa de reuso para este subconjunto ser bem inferior a todos os demais subconjuntos de reuso: 4,7%, contra 31,8% na configuração de Reuso Total. O reuso de Load e Store apresenta taxa de reuso de 7,7% e o subconjunto Add e Sub, taxa de 13,6%.

Analisando o subconjunto inverso dos desvios, ou seja, Tudo Menos Desvios, além do índice de eficiência, taxa de reuso e aceleração satisfatórios, quando comparados ao Reuso Total, não reusar desvios também garante um menor espaço para alocação de memória para as tabelas de reuso. Para as configurações das tabelas usadas nas simulações, obteve-se uma redução de 2,1% de espaço na Memo_Table_G e 10,7% em Memo_Table_T.

Dentro das avaliações de desempenho para o reuso de traços, o subconjunto Load e Store não poderia ser escolhido, uma vez que este subconjunto não faz reuso de traços; apenas de instruções isoladas. Isto decorre do fato de que na especificação do DTM e do RST as instruções de acesso à memória delimitam traços, de modo que não

seria possível duas instruções consecutivas neste subconjunto de reuso gerarem traços. Ainda assim, considerando somente instruções isoladas, este subconjunto possui índice de eficiência 2,29 vezes melhor do que no Reuso Total.

O índice de eficiência não necessariamente comprova que há um melhor subconjunto de reuso do que outro. Na realidade, todas as medidas utilizadas na análise servem como base para uma tomada de decisão, de acordo com aquilo que um projetista de hardware espere ganhar ou perder em função destas escolhas. Se o objetivo final é a obtenção de aceleração a qualquer custo, não há dúvidas de que o Reuso Total será a melhor escolha. Se há uma preocupação maior com a questão do consumo de energia, será necessário fazer o equilíbrio entre a taxa de reuso, a quantidade de acessos às tabelas de reuso e a aceleração encontrada em cada caso. Caso haja uma preocupação com o espaço alocado na memória, não reusar desvios pode representar uma economia de 12,8% em espaço físico.

É intuitivo perceber a importância de se diminuir o percentual de reuso, em detrimento da economia de energia. Se reusar os diversos subconjuntos isoladamente vai garantir uma redução da taxa de reuso, atingindo quase a totalidade da aceleração do Reuso Total, pensar em uma redução do domínio de instruções reusáveis pode ser uma alternativa interessante. Uma vez que o conjunto de reuso diminui, a quantidade de acessos às tabelas de reuso é reduzida. Como a redução dos acessos às tabelas de reuso acarreta diretamente um menor consumo de energia, repensar num subdomínio de instruções pode representar uma opção mais econômica, sem perdas relevantes de desempenho.

Este trabalho permitiu construir um panorama dos ganhos que podem ser obtidos através do mecanismo de reuso de traços, analisando todo o domínio de instruções reusáveis do DTM, dividido em subconjuntos de reuso. No momento em que os subconjuntos foram analisados, foi possível observar as peculiaridades de cada subconjunto, no que concerne aos ganhos obtidos com o mecanismo do reuso de traços. A partir das análises geradas por este trabalho, um projetista poderia elencar que ganhos ele deseja priorizar ao desenvolver uma arquitetura que implemente o reuso de traços.

Como foi possível observar nos resultados das experimentações, caso o objetivo principal do desenvolvimento de uma nova arquitetura seja o ganho em aceleração, é claro perceber que todo o domínio de instruções reusáveis deveria ser ativado para o mecanismo do reuso de traços. Uma vez que se considere como objetivo maior a redução do consumo de energia, o subconjunto Load e Store deveria ser o eleito, uma

vez que este não possui consumo de energia associado à leituras em Memo_Table_T. Isto ocorre porque esse subconjunto de reuso não faz reuso de traços, apenas de instruções isoladas. Neste cenário, se o projetista vislumbra como fator determinante um baixo consumo de energia, porém acredita ser fundamental que haja reuso de traços, a escolha pelo Subconjunto de Desvios seria a mais adequada, uma vez que apesar de possuir consumo de energia associado às leituras de Memo_Table_T e Memo_Table_G, o número de acessos a estas estruturas é cerca de 4,5 vezes menor do que na configuração de Reuso Total. Para uma arquitetura de hardware onde o espaço alocado em memória dedicado ao mecanismo de reuso é um ponto primordial, a escolha pelo subconjunto Load e Store também seria a opção mais adequada, uma vez que este subconjunto não necessita da estrutura de Memo_Table_T por não fazer reuso de traços. Tal particularidade deste subconjunto permite a economia de 45,6% em alocação de espaço em memória, se comparado à configuração de Reuso Total. Considerando mais uma vez ser primordial o reuso de traços, porém elegendo o subconjunto mais econômico no que concerne ao espaço de alocação, o subconjunto de reuso Tudo Menos Desvios poderia ser o escolhido. A exclusão dos campos específicos do reuso de desvios em Memo_Table_G e Memo_Table_T juntos correspondem a uma economia de 12,8% em alocação de espaço em memória. Por fim, o Índice de Eficiência, que foi criado durante as análises deste trabalho de dissertação, permite destacar o subconjunto de Desvios como o mais eficiente, uma vez que este apresenta a melhor relação de aceleração por percentual de instruções executadas, representando 90,2% do total de aceleração obtida no Reuso Total, utilizando apenas 19,6% de todas as instruções executadas.

Pelo fato da área de reuso de computação permitir uma gama muito abrangente de pesquisas, em suas diversas granularidades, vários trabalhos futuros podem ser realizados, partindo da idéia central deste trabalho:

- A avaliação realizada nos subconjuntos de instruções deste trabalho de dissertação pode ter seu foco também no consumo de energia, visando confirmar que subconjuntos de instruções dentro do domínio de instruções reusáveis podem, além de ter uma boa aceleração, manter uma baixa taxa de reuso, podem consumir o mínimo de energia sem perda de desempenho;
- Todas as simulações foram realizadas utilizando um preditor de dois níveis, que em geral apresenta bons resultados, devido à maior acurácia de sua predição. Um trabalho de observação do mecanismo de reuso de uma forma mais ampla, sem

estar relacionado somente aos previsores de dois níveis é importante para conhecer a influência dos diferentes tipos de previsores dentro do mecanismo de reuso;

- O mecanismo do reuso pode trazer vários benefícios diretos e indiretos no desempenho de uma arquitetura. Entender o impacto real que uma instrução pode causar para a computação de uma forma geral é uma tarefa de suma importância. Fazer análises mais profundas sobre o benefício que cada instrução reusada de forma isolada pode trazer pode possibilitar uma graduação em cada instrução na forma de atribuição de pesos, de modo que instruções com peso mais alto seriam mais valiosas para o reuso do que outras. Saber que instruções podem gerar correções de desvios ou redução dos acessos à memória cache ou ter o conhecimento de quais são as instruções que liberam mais unidades funcionais durante o seu reuso, por exemplo, podem representar atribuições de pesos diferentes a estas durante o processo de inclusão das mesmas nas tabelas de reuso. Estas atribuições podem posteriormente tornar-se uma política de substituição das entradas nas tabelas de reuso. Este conceito pode também ser estendido para o contexto de traços.

6. Referências Bibliográficas

AUSTIM, T.; LARSON, E.; ERNST, D. *SimpleScalar: an infrastructure for computer system modeling*. IEEE Computer, Los Alamitos, v.35, n.2, p.59-67, Feb. 2002.

BURGER, D., AUSTIM, T.M. *The SimpleScalar Tool Set Version 2.0,* Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Wisconsin, June 1997.

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 Traços 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 FILHO, E.M. *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.

GONZALEZ, A., TUBELLA, J., MOLINA, C. *The performance Potencial of Data Value Reuse*. Barcelona: Universitat Politécnica de Catalunya, 1998.

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

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.

JACOB, B.; MUDGE, T. *Notes on Calculating Computer Performance.* Technical Report CSE-TR-23195, Department of Electrical Engineering and Computer Science, University of Michigan, USA, March 1995.

MOLINA, C.; GONZÁLEZ, A.; TUBELLA, J. *Dynamic Removal of Redundant Computations.* In: ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING, 13., 1999, Rhodes. New York: ACM, 1999. P. 474–481.

HEWLETT-PACKARD DEVELOPMENT COMPANY, L.P. (2008). HP Labs: CACTI. Acesso em 1 de Março de 2009, disponível em HP Labs: http://www.hpl.hp.com/research/cacti/

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

RICHARDSON, S.E. *Caching Function Results: fast arithmetic by avoiding unnecessary computation.* California: Sun Microsystems Laboratories, 1992. Technical Report. (Tech Report SMLI TR-92-1).

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 AnnualInternational 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.

SODANI, A., SOHI, G.S. *An Empirical Analysis of Instruction Repetition.* In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp 35-45, San Jose, October 1998.

YANG, J.; GUPTA, R. *Load Redundancy Removal through Instruction Reuse*. In: ANNUAL INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE, 24., 1991, Albuquerque. Los Alamitos: IEEE Computer Society, 2000. p.61-68.

7. ANEXO A

Tabela A.1 – Aceleração Add e Sub – Valores Encontrados

Benchmarks	*Add e Sub*	*Tudo Menos Add e Sub*	*Reuso Total*
Bzip	1,148465054	1,191760717	1,200219079
Gcc	1,119499410	1,159631012	1,227496310
Gzip	1,047966267	1,183744063	1,101940902
Mcf	1,006352368	1,037134839	1,033232270
Parser	1,144060039	1,182398275	1,289933588
Vortex	1,050899074	1,127406293	1,190209796
MH	1,083512272	1,144352372	1,167634203

Tabela A.2 – Aceleração Desvios – Valores Encontrados

Benchmarks	*Desvios*	*Tudo Menos Desvios*	*Reuso Total*
Bzip	1,047778028	1,217124985	1,200219079
Gcc	1,072500638	1,168599239	1,227496310
Gzip	1,036237595	1,098434034	1,101940902
Mcf	0,999290931	1,027760869	1,033232270
Parser	1,067764016	1,228863358	1,289933588
Vortex	1,111300148	1,095594955	1,190209796
MH	1,054681056	1,134812771	1,167634203

Tabela A.3 – Aceleração Load Store – Valores Encontrados

Benchmarks	*Load e Store*	*Tudo Menos Load e Store*	*Reuso Total*
Bzip	1,018541736	1,105110556	1,200219079
Gcc	1,030494524	1,242889731	1,227496310
Gzip	1,025593283	1,084368872	1,101940902
Mcf	1,011292438	1,027169887	1,033232270
Parser	1,025547302	1,276501559	1,289933588
Vortex	1,021980527	1,205499268	1,190209796
MH	1,022205010	1,149837070	1,167634203

Tabela A.4 – Taxa de Reuso Add e Sub

Benchmarks	*Add e Sub*	*Tudo Menos Add e Sub*	*Reuso Total*
Bzip	0,1371	0,1416	0,2610
Gcc	0,1768	0,1851	0,3666
Gzip	0,1478	0,1888	0,3298
Mcf	0,1019	0,1634	0,2741
Parser	0,1492	0,2198	0,4086
Vortex	0,1267	0,1670	0,3130
MH	0,1360	0,1743	0,3178

Tabela A.5 – Taxa de Reuso Desvios

Benchmarks	*Desvios*	*Tudo Menos Desvios*	*Reuso Total*
Bzip	0,0269	0,2817	0,2610
Gcc	0,0443	0,3270	0,3666
Gzip	0,0331	0,3131	0,3298
Mcf	0,0830	0,1864	0,2741
Parser	0,0849	0,3140	0,4086
Vortex	0,0616	0,2306	0,3130
MH	0,0461	0,2645	0,3178

Tabela A.6 – Taxa de Reuso Load e Store

Benchmarks	*Load e Store*	*Tudo Menos Load e Store*	*Reuso Total*
Bzip	0,0701	0,1568	0,2610
Gcc	0,1039	0,2561	0,3666
Gzip	0,1081	0,2341	0,3298
Mcf	0,0514	0,2316	0,2741
Parser	0,0848	0,2937	0,4086
Vortex	0,0755	0,2207	0,3130
MH	0,0773	0,2238	0,3178

Tabela A.7 – Eficiência Add e Sub

Benchmarks	*Add e Sub*	*Tudo Menos Add e Sub*	*Reuso Total*
Bzip	2,24	2,45	1,20
Gcc	2,89	1,91	1,24
Gzip	2,06	2,41	1,10
Mcf	2,76	1,65	1,04
Parser	2,87	2,02	1,31
Vortex	3,93	1,55	1,19
MH	2,68	1,94	1,17

Tabela A.8 – Eficiência Desvios

Benchmarks	*Desvios*	*Tudo Menos Desvios*	*Reuso Total*
Bzip	7,26	1,43	1,20
Gcc	5,37	2,64	1,24
Gzip	6,76	3,37	1,10
Mcf	3,58	2,95	1,04
Parser	4,69	2,94	1,31
Vortex	6,73	1,82	1,19
MH	5,39	2,30	1,17

Tabela A.9 – Eficiência Load Store

Benchmarks	*Load e Store*	*Tudo Menos Load e Store*	*Reuso Total*
Bzip	3,10	1,65	1,20
Gcc	2,64	2,06	1,24
Gzip	3,37	1,56	1,10
Mcf	2,95	1,58	1,04
Parser	2,94	2,01	1,31
Vortex	1,82	2,76	1,19
MH	2,70	1,86	1,17