EXPLORANDO DINAMICAMENTE O REUSO DE TRACES EM NÍVEL DE ARQUITETURA DE PROCESSADOR

Amarildo Teodoro da Costa

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

Aprovada por:

Prof. Felipe Maia Galvão França, Ph.D.

Prof. Edil Severiano Tavares Fernandes, Ph.D.

Prof. Cláudio Luis de Amorim, Ph.D.

Prof. Eugene Francis Vinod Rebello, Ph.D.

Prof. Alberto Ferreira de Souza, Ph.D.

Prof. Philippe Olivier Alexandre Navaux, Dr. Ing.

Prof. Siang Wun Song, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

ABRIL DE 2001

COSTA, AMARILDO TEODORO DA

Explorando Dinamicamente o Reuso de Traces em Nível de Arquitetura de Processador [Rio de Janeiro] 2001

XVII, 181 pp., 29.7 cm, (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 2001)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1 – Reuso Dinâmico de Traces

2 – Arquitetura de Processador

I. COPPE/UFRJ II. Título (série)

DEDICATÓRIA

À minha esposa e meus filhos,

exemplos de amor, união e luta

AGRADECIMENTOS

Aos professores Felipe Maia Galvão França e Eliseu Monteiro Chaves Filho, que me orientaram competentemente e participaram ativamente deste trabalho de tese. Foram fundamentais as discussões que travamos em torno do trabalho, e as reuniões que determinaram muitas soluções e direções a serem tomadas. Felizmente, tive apoio e incentivo constante do professor Felipe, para poder explorar e desenvolver livremente, novas idéias e experiências que culminaram neste trabalho de tese. Felizmente também, tive o suporte constante do professor Eliseu, para lapidar as idéias e realizá-las de forma consistente e rigorosa. Certamente, o aprendizado decorrente do convívio com estes professores, permanecerá ativo durante toda a minha vida profissional.

À Universidade Federal do Rio de Janeiro e, em especial ao Programa de Engenharia de Sistemas e Computação, por ter contribuído para minha formação.

Ao Instituto Militar de Engenharia e, em especial ao Departamento de Engenharia Eletrônica, pela oportunidade de aperfeiçoar meus estudos, galgando assim, mais um degrau da carreira docente e da pesquisa científica.

Aos professores Edil S. T. Fernandes, Inês de Castro Dutra, Vitor Santos Costa, Ricardo Bianchini, Claúdio Luís de Amorim, Eugene F. V. Rebello, Alberto Ferreira de Souza, Phillipe O. A. Navaux e Siang Wun Song, pela atenção dispensada.

Aos colegas de curso da COPPE/Sistemas, Zhijum Yang, Silvio Tadeu Canola, Enrique Vinicius C. Erazo, Álvaro da Silva Ferreira, Gabriel Pereira da Silva e Luís Maltar Castello Branco, que direta ou indiretamente me ajudaram com suas opiniões e discussões sobre diversos temas ligados à ciência da computação. Agradecimento em especial à Luíz Márcio de Aquino Viana, que identificou alguns pontos em

aberto na implementação do mecanismo DTM, e que discutiu inúmeras situações e possibilidades de aperfeiçoamento do mecanismo.

Aos meus amigos, Reinaldo Teixeira e Edson Nazareno S. de Souza, pelo apoio e encorajamento constantes durante o curso.

À minha esposa Zilnete e meus filhos Brunna e Allan, pelo carinho, compreensão e incentivo constante, para enfrentar os obstáculos pessoais e profissionais.

Ao meu pai, Sr. Almir Dias da Costa, que sempre me incentivou a trilhar o caminho do saber.

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

EXPLORANDO DINAMICAMENTE O REUSO DE TRACES EM NÍVEL DE ARQUITETURA DE PROCESSADOR

Amarildo Teodoro da Costa

Abril/2001

Orientadores: Felipe Maia Galvão França
Eliseu Monteiro Chaves Filho

Programa: Engenharia de Sistemas e Computação

Este trabalho introduz e explora o conceito de *Memorização* e *Reuso Dinâmico de Traces*. Este conceito é avaliado através de um mecanismo concebido para tal propósito e denominado *Dynamic Trace Memoization* (*DTM*). Este mecanismo é implementado em nível de arquitetura de processador e, é capaz de explorar reuso sobre computações redundantes com granularidade em nível de *traces* (sequência de instruções dinâmicas). Ao identificar um trace redundante, o mecanismo *DTM* desconsidera a execução de todas instruções cobertas pelo trace, evitando deste modo a reexecução destas. Experimentos realizados, considerando o simulador de um processador superescalar suportando execução fora de ordem e utilizando como entrada os programas de teste do *SPEC95*, identificaram uma grande quantidade de redundância que pôde ser reusada. A exploração da redundância identificada, provocou reduções: no número de instruções executadas; dos caminhos críticos determinados por dependências verdadeiras; do número de desvios preditos incorretamente e do número de acessos feitos aos caches. Conseqüentemente, sua aplicação provocou expressivos ganhos de performance no processador superescalar que o incorporou.

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

EXPLOITING THE REUSE OF TRACES DYNAMICALLY AT THE PROCESSOR ARCHITECTURE LEVEL

Amarildo Teodoro da Costa

April/2001

Advisors: Felipe Maia Galvão França
Eliseu Monteiro Chaves Filho

Department: Computing and Systems Engineering

This work introduces and exploits the concept of *Memoization and Dynamic Reuse of Traces*. This concept is evaluated through a mechanism called *Dynamic Trace Memoization* (*DTM*). This mechanism is implemented at the processor architecture level, and is capable of exploiting reuse on redundant computations with granularity at the trace level (sequence of dynamic instructions). When identifying redundant traces, the *DTM* mechanism skips the execution of all instructions covered by the trace, thus preventing the re-execution of these instructions. Experiments have been carried out using the *SPEC95* program as input to a simulator of a superscalar processor which supports out-of-order execution. These experiments identified a large amount of redundancy that could be reused. Exploiting the identified redundancy (through the reuse), cause reductions: in the number of executed instructions; in the critical paths created by true dependences; in the number of incorrectly predicted branches and in the number of cache accesses. Consequently, its application produces significant benefits in performance for the (superscalar) processor architectures which incorporate it.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Processadores mais poderosos são cada vez mais requeridos. Esta demanda se faz necessária, para suportar os requisitos crescentes de programas de aplicação mais sofisticados e diversificados, sejam estes científicos, comerciais ou de lazer. Em um processador, são alcançados crescentes níveis de desempenho atuando-se em:

- Sua confecção física, ou seja, na tecnologia dos semicondutores, de modo a produzir processadores funcionando em elevadas freqüências de clock;

- Sua arquitetura, ou seja, através da inclusão e aperfeiçoamento de mecanismos que permitam executar um maior número de instruções a cada ciclo de clock.

Pesquisas recentes na área de arquitetura de processadores tem buscado prover condições, que permitam a execução paralela do maior número possível de instruções por ciclo de clock. Para atingir este objetivo, os processadores atuais incorporam diversos mecanismos que permitem explorar cada vez mais o paralelismo temporal/espacial existente em programas. Como forma de contribuir direta ou indiretamente com este objetivo, novas técnicas estão sendo investigadas, dentre elas, as primeiras observações sobre localidade e redundância de valores [45, 44, 63, 35, 20, 30]. Estas recentes pesquisas, identificaram que durante a execução de programas, um grande percentual de instruções fazem referências constantes aos mesmos valores a elas previamente instanciados. Uma nova visualização destes conceitos, alavancaram novas direções e esforços de pesquisa, produzindo variadas investigações científicas com o intuito de explorar este comportamento através de especulação e reuso.

A recorrência à valores de entrada já observados é definida como *localidade tem-*

poral/espacial de valores. Esta característica descreve um comportamento possivelmente redundante de uma computação e a previsibilidade dos resultados a serem produzidos, visto que computações que apresentam as mesmas entradas, podem produzir os mesmos resultados. O reuso de computações redundantes [1], reapresenta-se em um novo cenário, como uma das novas linhas de investigação que podem efetivar ganhos de performance em processadores, visto que explorar este comportamento equivale a reduzir o número de instruções (que compõe a computação redundante) que são executadas por programas.

Computações redundantes exploradas de forma estática ou dinâmica, são suportadas pela técnica de memorização [48]. Explorar o reuso de uma computação redundante por intermédio de memorização não é um conceito recente. Entretanto, a aplicação deste conceito em nível de arquitetura de processadores, revelou novos horizontes de investigações promissoras. Desafiadoramente, o cenário de aplicação deste conceito tem sido deslocado para uma identificação dinâmica de tais computações, o que acrescenta diferentes considerações e um alto grau de complexidade, a começar pelas diferentes possibilidades, comportamento e granularidades das computações redundantes. Uma outra consideração não menos complexa se refere às funções que o mecanismo em hardware deve acumular: identificação de redundância, memorização e reuso. Estas tarefas devem ser executadas de forma harmoniosa com os outros mecanismos incorporados ao processador, de modo a não introduzir inconsistências, atrasos, adição de carga a outros elementos funcionais, etc ....

A proposição de novos mecanismos para explorar computações redundantes através de reuso e propostas para sua incorporação em processadores, baseiam-se na premissa de que *reusar uma computação equivale a não reexecutá-la*. Esta situação ocorre quando uma instância de uma dada computação, faz referência a uma instância previamente executada ou também repetida (nestes casos, já memorizada).

Como exemplo de se explorar o reuso de computações redundantes, a Figura 1.1, apresenta comparativamente, diferentes modelos de execução. A Figura 1.1(a), apresenta um trecho de código a ser executado múltiplas vezes. As respectivas execuções

[1] Computação redundante aqui mencionada, pretende generalizar as diferentes e possíveis granularidades: funções, trechos de código estático, traces, blocos básicos, instruções, etc ...

considerando situações ideais para cada cenário são apresentadas: a Figura 1.1(b), uma arquitetura seqüencial explorando paralelismo temporal; a Figura 1.1(c), uma arquitetura superescalar explorando paralelismo espacial e temporal; e finalmente a Figura 1.1(d), uma arquitetura explorando o reuso da computação descrita pelo trecho de código (supondo redundante). Esta última, demonstra que uma computação redundante possuindo um número n de instruções, ao ser reusada, equivale a n instruções executadas ao mesmo tempo. Esta redução, provoca a elevação do número de instruções executadas por unidade de tempo.

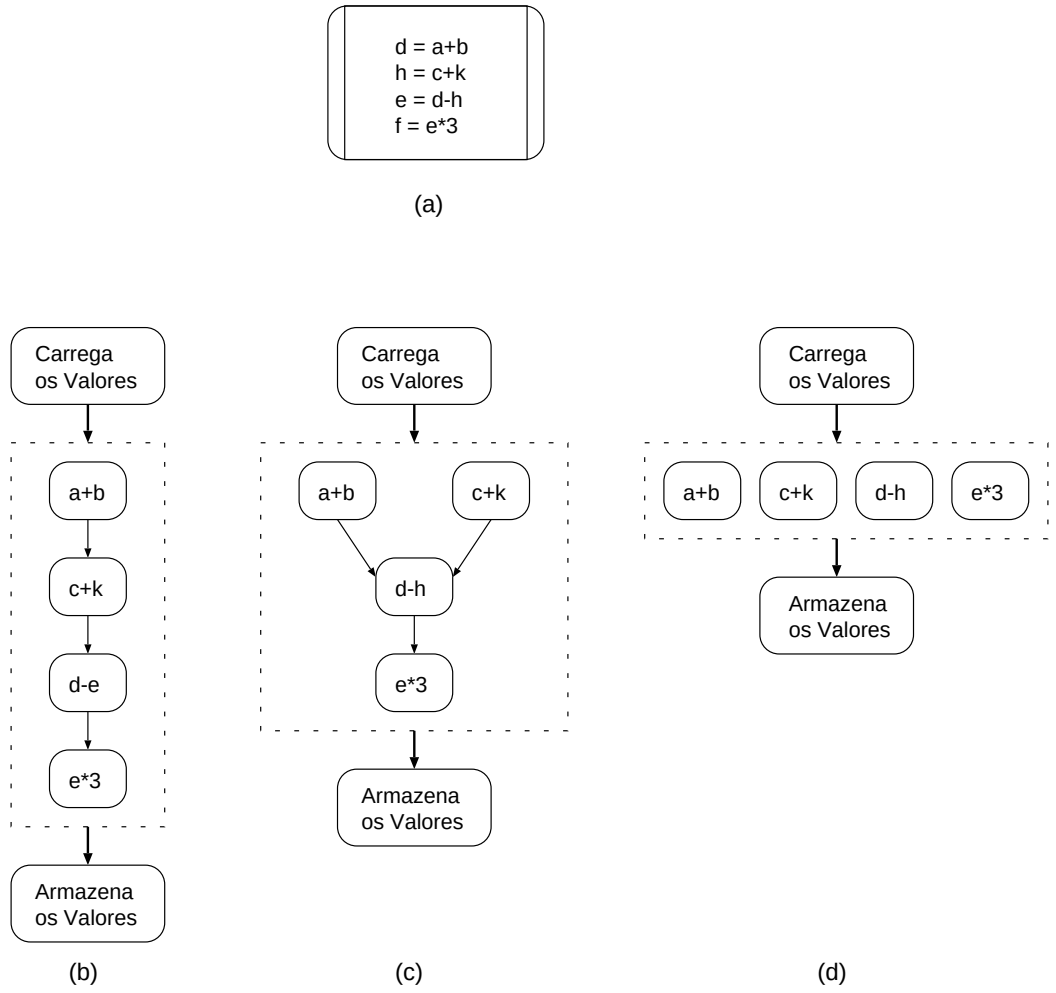


Figura 1.1: Caracterização de cada cenário considerando situações ideais: (a) seqüência de código, (b) arquitetura escalar, (c) arquitetura superescalar, (d) arquitetura explorando reuso.

A incorporação de mecanismos para explorar o reuso de computações redundantes com diferentes granularidades e em nível de arquitetura de processadores,

apresenta-se com um grande potencial que irá influenciar as futuras gerações de processadores. Este trabalho de tese pretende contribuir neste sentido, avaliando e propondo modos de identificação, memorização e reuso de computações redundantes com granularidade em nível de traces (seqüências de instruções dinâmicas redundantes).

1.1 Computação redundante e classificação dos mecanismos para exploração de redundância

Computação Redundante ou Repetitiva, é caracterizada pela reexecução de uma dada computação que foi previamente executada com os mesmos operandos de entrada, e conseqüentemente produzindo os mesmos resultados de saída. Explorar a redundância de uma computação equivale a se reusar a computação, evitando-se deste modo a reexecução da mesma, e minimizando assim, o número de computações que devam ser executadas para que um dado programa seja completamente executado. Para realizar este procedimento faz-se necessário: armazenar previamente os resulatados das computações efetuadas (após seleção), identificar em que oportunidades estas possam ser reusadas, e reusá-las.

A seguir será apresentada uma caracterização das fases necessárias para explorar o reuso de computações redundantes. Cada fase possuirá internamente uma caracterização distinta e decorrente das considerações expostas. Esta caracterização, pretende facilitar o entendimento das diferentes aproximações que serão avaliadas e propõe uma classificação dos mecanismos para exploração de reuso. Esta classificação será convencionada e referenciada no decorrer deste trabalho.

1. **Identificação de uma computação redundante:**

   - *Estaticamente*

     A computação é identificada como redundante, observando-se por inspeção do código ou durante a compilação, sua natureza recorrente. A partir desta identificação e para uma eventual utilização futura, é feita a marcação do trecho de código correspondente a computação. O trecho

marcado sempre será avaliado com relação ao seu provável reuso.

- *Dinamicamente*

  A computação é identificada como redundante por inspeção de suas entradas e valores a elas atribuído. A identificação é feita durante a execução do programa que a contém e sem marcações efetuadas a priori.

2. **Memorização da computação:**

   - *Software*

     As instâncias da computação candidata a reuso são armazenadas em estruturas de dados inclusas no programa que contém a dada computação.

   - *Hardware*

     As instâncias da computação candidata a reuso são armazenadas diretamente em estruturas de memorização pertencentes ao hardware do mecanismo de exploração de reuso.

3. **Identificação das oportunidades de reuso:**

   - *Software*

     O trecho de código correspondente a computação, é marcado a priori como candidato à memorização, e será verificado oportunamente, antes de sua execução, através de um mecanismo de exploração de reuso implementado em software.

   - *Hardware*

     O trecho de código correspondente a computação, deverá ser identificado por intermédio de características que o relacione à sua execução (Ex: endereço de memória, código de operação, etc ...), ou por intermédio de heurísticas inclusas no mecanismo de exploração de reuso implementado em hardware.

A partir do que foi estabelecido, pode-se classificar as diferentes estratégias de exploração de reuso que serão expostas no decorrer do texto.
Será utilizada a seguinte convenção: *ID-X/ME-Y/OP-Y* - Onde:

- *ID*- corresponde a identificação de uma computação redundante.

- *ME*- corresponde a memorização da computação.
- *OP*- corresponde a identificação das oportunidades de reuso.
- *X* - será instanciado, com *E* para estático e *D* para dinâmico.
- *Y* - será instanciado, com *S* para software e *H* para hardware.

Exemplo:

ID-E/ME-H/OP-S significa que: a identificação de uma computação redundante é feita estaticamente; a memorização da computação é feita diretamente no hardware; e a identificação das oportunidades de reuso é feita por software.

Em particular:

ID-E/ME-S/OP-S - será referenciado como Memorização Estática;

ID-D/ME-(S ou H)/OP-H - será referenciado como Memorização Dinâmica.

1.2 Histórico

Até o início da década de 90, a computação redundante foi quase que exclusivamente explorada através de *Memorização Estática*. Nesta, uma dada computação com comportamento repetitivo era identificada estaticamente, observando-se seu comportamento recorrente.

A técnica de *Memorização Estática* aplicada a um dado programa, consiste na adição de um trecho de código, sendo este responsável pelo armazenamento (em uma tabela) das computações candidatas a reuso (previamente selecionadas), e pela identificação e reuso de tais computações. O armazenamento das computações candidatas à reuso, é feito através do tabelamento de um identificador da computação, seus operandos de entrada e operandos de saída (resultados produzidos pela computação). A exploração de reuso de uma computação é efetuada quando uma instância da computação candidata à reuso é identificada como presente na tabela. A partir da identificação, evita-se a reexecução da computação, atualizando-se apenas os operandos de saída (resultados). A computação redundante sempre foi até então, considerada como um elemento a ser explorado pela memorização estática, ou seja,

uma característica comportamental de um programa, visível ao programador e por ele implementada.

O conceito de memorização foi introduzido em [48]. A idéia básica é salvar em uma tabela, as entradas usadas e os resultados produzidos (conjuntamente) por funções livres de efeitos colaterais, e reusar os resultados das funções para os casos em que suas respectivas entradas sejam repetitivas, ou seja, identificadas como presentes na tabela de memorização. O mecanismo de reuso consiste em tabelas implementadas em estruturas de dados e alocadas no próprio programa, e com os procedimentos de busca e atualização, efetuadas em alto nível por software (incluso no programa). Desde sua concepção, este procedimento tem sido usado, principalmente no contexto de linguagens declarativas como Prolog, Lisp e ML [4]. A Figura 1.2, exemplifica a seqüência de eventos efetuados para verificar se a função *fat(n)* (fatorial) possui um resultado já armazenado na tabela de memorização (a função *fat(n)* foi escolhida para ser memorizada, devido ao seu comportamento recorrente).

O procedimento de reuso é resumido nos seguintes passos:

(i) Para cada execução de fat(n), esta será instanciada com um determinado valor de entrada ($n = x$);

(ii) Antes da execução de fat(x), será executado um procedimento de busca à tabela de memorização, utilizando-se a atual instância fat(x) como chave de busca;

(iii) A tabela será varrida pesquisando-se por uma entrada que identifique uma execução prévia de fat(x);

(iv) Se a entrada procurada em (iii) existir, então será retornado o valor resultante e já armazenado da execução de fat(x); senão, a função fat(x) será executada normalmente e o resultado produzido será inserido em alguma entrada da tabela de memorização (para que em uma próxima execução da função com o mesmo valor de entrada, esta não necessite ser executada).

Por muitos anos, as técnicas de exploração de computações redundantes limitaram-se à *Memorização Estática*. Este tipo de memorização restringe seus escopos de

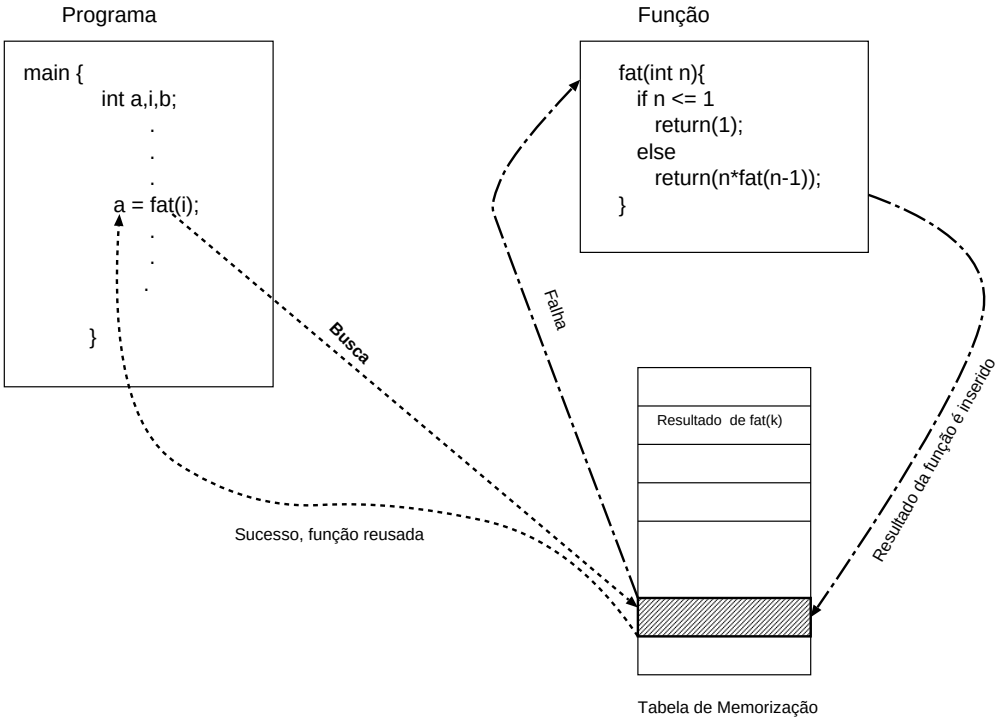


Figura 1.2: Memorização de uma função redundante.

aplicação à regiões de código em alto nível e sem efeitos colaterais. Estas regiões de código são representadas por funções, procedimentos e trechos de código em alto nível. As regiões de código selecionadas para aplicação de *Memorização Estática*, são analisados predominantemente de forma estática, ou seja, identifica-se a priori se tais regiões são redundantes, considerando-se apenas a natureza recorrente dos problemas algorítmicos que as mesmas representam. Atualmente, vários estudos em curso [14, 42, 28, 12], direta ou indiretamente, procuram reduzir o efeito limitante da análise estática, por esta ainda representar uma característica inibitória à memorização e à identificação de computações redundantes. Entre outros trabalhos, [46] explora redundância, procurando formas automatizadas para identificá-las em linguagens funcionais e imperativas.

Apesar de não ser classificada como *Memorização*, em [6], foi proposta a utilização de tabelas previamente preenchidas com resultados parciais de operações de divisão. Este procedimento possui como objetivo, diminuir a latência de unidades funcionais especializadas em operações de divisão e multiplicação, efetuando para isto, um acesso à uma tabela, para obtenção de resultados parciais destas operações.

Este procedimento não é caracterizado como *Memorização*, pois o mecanismo proposto é pré-definido, e por conseguinte, não armazena e não identifica computações redundantes. Versões relacionadas a este mecanismo, foram e ainda são implementadas em processadores atuais. Curiosamente, um erro na transposição de valores para esta tabela utilizada pelo mecanismo, foi responsável pelo famoso erro na unidade de divisão do Pentium [11].

Em [32], foi efetuada uma das primeiras pesquisas explorando uma variação de memorização e sua aplicação em nível de arquitetura de processadores. Neste trabalho, foi apresentada e simulada uma proposta de memorização *ID-D/ME-H/OP-H*. Esta, possuía como objetivo, eliminar dinamicamente subexpressões redundantes [2, 9] que são identificadas apenas em tempo de execução. O mecanismo original era aplicado a computações denominadas *frases*, e o processador alvo possuía um conjunto de instruções com execução orientada à pilha.

Uma contribuição extremamente importante foi apresentada em [53]. Este, baseado principalmente nos trabalhos [48, 32], aplica o uso de memorização *ID-E/ME-S/OP-S* à programas científicos e avalia seu efeito na performance de processadores escalares. Aliado a esta avaliação e com o intuito de reusar operações repetitivas, no mesmo trabalho [53], é proposta a aplicação de memorização *ID-D/ME-H/OP-H* à instruções com altas latências de execução (multiplicação, divisão e raiz quadrada).

A partir do trabalho [53], em [50] foi aplicada memorização *ID-D/ME-H/OP-H* em nível arquitetural, mais especificamente às unidades funcionais de divisão, com o objetivo de reduzir a latência destas unidades. Utilizaram para este fim, uma pequena memória cache associada a unidade funcional em questão. Em [18], foi avaliada uma variante deste método, visando reduzir a latência de unidades funcionais que efetuam operações com altas latências de execução; e foi investigado também, o impacto desta técnica na aceleração de *Benchmarks Multi-Media*. Os trabalhos [53, 50, 18], diferem-se ligeiramente entre si, com relação à localização da tabela de memorização, e variações do modo de identificação de redundância e posterior reuso.

Os trabalhos [45, 29, 44], introduziram o conceito de localidade de valores e predição de valores. Basicamente, a predição de valores é calcada na recorrência a valores previamente instanciados ao elemento alvo da predição (uma posição de

memória ou um registrador associado à uma instrução). A escolha do valor a ser predito, considera os n últimos valores instanciados a estes elementos. Em contrapartida às técnicas convencionais de memorização, a predição de valores considera a previsibilidade de valores que podem ser instanciados à uma instrução, e explora este comportamento de forma especulativa.

O trabalho [52], procurou avaliar o efeito da memorização aplicada a funções codificadas em alto nível. Neste trabalho, o tabelamento das computações redundantes (funções) é suportado diretamente por elementos de memorização implementados em hardware (caches). Os procedimentos responsáveis pela identificação de redundância e posterior reuso, são de responsabilidade exclusiva de instruções especializadas do processador e incluídas para este fim. As instruções especializadas são inseridas a priori no código objeto do programa alvo, de modo a contornar as funções a serem avaliadas. A partir do convencionado anteriormente, pode-se classificar tal esquema como sendo *ID-E/ME-H/OP-H*.

Em [63], foi introduzido o conceito de reuso de instruções dinâmicas. Neste, é efetuada a exploração de computação redundante com granularidade em nível de instruções dinâmicas e cadeias de instruções dinâmicas dependentes. A técnica de memorização *ID-D/ME-H/OP-H*, é implementada através de três esquemas de reuso e com escopo de aplicação limitado à fronteiras de blocos básicos. Basicamente, dois dentre os esquemas de reuso apresentados neste trabalho reeditam as técnicas utilizadas em [32], diferindo apenas nas adaptações do mecanismo à arquitetura do processador alvo. Ainda neste trabalho, os autores incorporaram os esquemas de reuso propostos em processadores superescalares e distinguiram dois tipos de reuso. Os tipos de reuso identificados são decorrentes do modo de identificação de instruções redundantes, o *General Reuse* e o *Squash Reuse*. O *General Reuse* identifica e explora o reuso realmente existente na execução de um programa; enquanto o *Squash Reuse* explora a redundância decorrente da execução especulativa de trechos de um programa, ou seja, promove uma redundância artificial ao programa quando fracassa a confirmação de correção de um caminho especulativamente escolhido (predito).

Em [64], foi feito um estudo empírico das causas da repetição de instruções em programas. Neste trabalho, os autores analisaram a redundância de instruções dinâmicas considerando diferentes níveis de contextos: globais (programas) e locais

(funções).

Em [65], foi feito um estudo comparativo entre predição de valores e reuso de instruções. Neste trabalho, o esquema de cadeias dependentes apresentado em [63], é estendido e simulado para prover uma base de comparação aos esquemas de predição de valores.

Em [35], foi apresentada a exploração de computação redundante com granularidade em nível de blocos básicos. Neste trabalho foi utilizada a técnica de Memorização *ID-D/ME-H/OP-H* (apesar de alguns blocos básicos serem identificados estaticamente, outros só podem ser identificados dinamicamente).

Em [20], foi apresentada uma alternativa ao reuso em nível de instruções simples e em nível de blocos básicos. Neste trabalho, os autores introduzem o conceito de *Memorização Dinâmica de Traces* e avaliam seu potencial de reuso. Nele foram consideradas tabelas de memorização com um número finito de entradas e a utilização de memorização *ID-D/ME-H/OP-H*, onde seqüências de instruções dinâmicas (incluindo instruções de desvio) são memorizadas e reusadas dinamicamente pelo hardware e, quando identificadas como redundantes, são reusadas por completo. Este trabalho praticamente generaliza os trabalhos [63, 35] e busca explorar o reuso sem limitar as regiões de análise. Estender a exploração de reuso atravessando blocos básicos, foi baseado nas mesmas considerações sobre a limitação de paralelismo em programas e nas propostas para aumentar a exposição de *ILP*, ou seja, ampliando a região de análise [54, 26, 43].

Em [49], foi apresentado um mecanismo baseado em memorização *ID-D/ME-H/OP-H*, com o objetivo de explorar computação redundante em nível de instrução. Neste trabalho são apresentadas comparações de desempenho ao mecanismo apresentado em [63] e a uma extensão do mecanismo proposto em [53].

Em [30], foi avaliado o reuso de traces contendo instruções genéricas (inclusive instruções com efeitos colaterais). Neste trabalho foi feita uma avaliação do potencial desta técnica, considerando um processador ideal e hardware de memorização ilimitado.

Em [19], foi apresentada uma combinação entre software e hardware, para identificar e explorar o reuso de computações redundantes. Este trabalho aplica técnicas de compilação a partir do *profile* do programa alvo, com o objetivo de identificar es-

taticamente as regiões de código com potencial de redundância. Sendo identificadas as regiões de código, estas são marcadas e, o controle de tabelamento, identificação e reuso, correm por responsabilidade de um mecanismo em hardware (*ID-E/ME-H/OP-H*).

Em [22, 23, 24] é proposta e avaliada uma implementação do mecanismo descrito em [20] e [21]. O mecanismo proposto é incorporado a um processador superescalar e: são efetuadas avaliações de redundância; ganhos de performance; e recursos arquiteturais críticos que são comuns ao mecanismo e ao processador.

Em [36] é apresentado um refinamento de [35]. Neste trabalho, os blocos básicos são subdivididos em sub-blocos (através de heurísticas). Esta subdivisão consegue contornar algumas deficiências decorrentes do reuso de blocos básicos e promove uma maior localidade de valores.

1.3 Motivações

Como já exposto, computação redundante (ou repetitiva) é caracterizada por computações que apresentam operandos de entrada repetidos (já observados em execuções anteriores da mesma computação) e, conseqüentemente repetição dos valores de saída produzidos pela computação. A exploração de reuso de funções com comportamento redundante, acelera eficientemente a execução de programas, pois evita-se a reexecução destas e, conseqüentemente, de todas as instruções que a compõe. Basicamente, neste cenário explora-se a redundância implementando-se um tabelamento das ativações das funções propensas a repetição (recorrência, invariabilidade de valores de entrada, etc ...), ou seja, armazena-se informações sobre: a função; os valores atribuídos (direta ou indiretamente) aos seus parâmetros de entrada; e os valores de saída produzidos e decorrentes da execução. Para cada ativação da função, é efetuada uma busca na tabela de memorização com o objetivo de identificar uma ativação prévia e idêntica à atual. Se tal evento ocorrer, então a função não será executada pois seus valores de saída serão obtidos da tabela de memorização.

A função que representa a conjectura de Collatz, na Figura 1.3(a), apresenta em sua codificação, recorrência a suas próprias execuções prévias (recursividade). Para o exemplo apresentado, o tabelamento das ativações da função, permite que esta

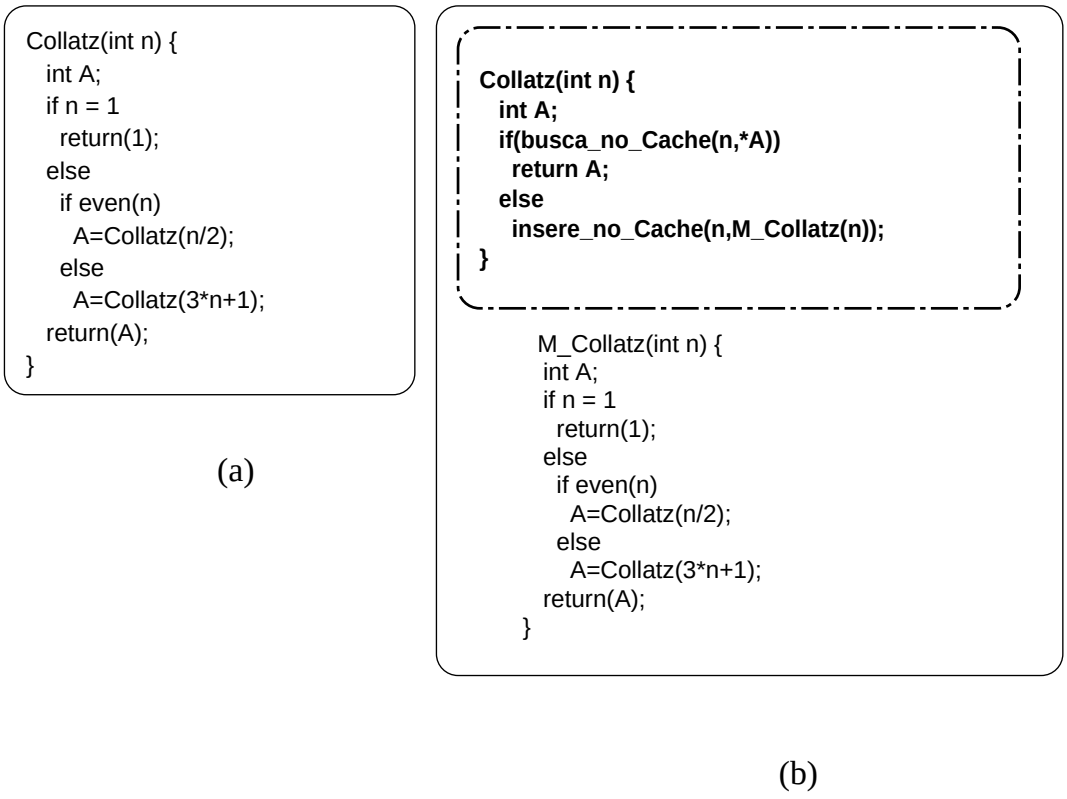


Figura 1.3: Aplicando memorização, (a)função original, (b)função implementando memorização estática.

seja reusada; evitando assim sua reexecução e reduzindo o número de instruções que devem ser executadas. Na Figura 1.3(b) é apresentada uma versão da função original incluindo memorização.

Observar e reconhecer em programas, se uma função ou procedimento possui uma característica redundante e em que situações isto pode ocorrer, pode não ser uma tarefa trivial. Além do exposto, há de se considerar ainda os efeitos colaterais que possam ocorrer internamente à função. Estes, podem não serem explicitamente visíveis no código estático; ou visíveis, porém não podem ser tratados facilmente (ex: uma instrução que altere uma variável global, uma instrução de E/S, um ponteiro, etc ...), e conseqüentemente afetam a correção do programa.

Além do estabelecido anteriormente, a exploração de redundância através de memorização estática pode apresentar resultados negativos e, conseqüentemente, uma sobrecarga maior de processamento. Isto ocorre para os casos onde os procedimentos ou funções selecionadas para aplicação da técnica, não apresentem um grau de redundância satisfatória e exposta à exploração, de modo a compensar o

acréscimo de execução da memorização (inclusões e pesquisas feitas por software na tabela de armazenamento). O escopo de aplicação desta técnica é restrita a certos procedimentos e funções que possuem uma peculiaridade de recorrência. Este tipo de memorização avaliado, apresenta-se extremamente eficiente para explorar computações redundantes quando estas são imediatamente observáveis. Entretanto, pode-se considerar que este tipo de exploração está situada em nível de software e é aplicável apenas à programas codificados em alto nível.

A maior motivação para este trabalho de tese foi decorrente da complexidade de se identificar computações redundantes de forma estática. Além da dificuldade de inspeção do código fonte de um programa, outra dificuldade não menos complexa, recai na identificação dos efeitos colaterais que possam estar inclusos nas computações selecionadas e que impedem a correção da aplicação de memorização. As discussões adiante, identificam e discriminam os pontos críticos que motivaram o desenvolvimento deste trabalho de tese e as soluções adotadas.

Os resultados provenientes dos trabalhos [53, 52], formaram a base de investigação deste trabalho de tese. A partir de observações sobre as restrições impostas pelos mesmos, as seguintes questões foram propostas:

(i) Existe redundância em funções ou trechos de código que não possuam características recorrentes?

(ii) Pode-se contornar os efeitos colaterais que impedem a exploração de reuso em trechos de código redundantes?

(iii) De que modo pode-se abolir a seleção estática de trechos de código candidatos a reuso?

Inicialmente, para responder tais questões, optou-se por uma análise de redundância a partir dos elementos atômicos, ou seja, instruções. Uma análise do comportamento redundante de instruções dinâmicas (sendo estas avaliadas isoladamente), pôde ser inferido a partir dos experimentos de [45, 29, 44]. Nestes trabalhos, é verificada uma alta localidade de valores para um grande percentual de instruções dinamicamente capturadas (durante a execução de programas de teste), ou seja, observou-se que muitas destas instruções são repetidamente instanciadas

com valores de entrada já observados. Posteriormente em [63, 57, 17], esta localidade temporal de valores é confirmada, sendo que foi explorada em [63], através de *Memorização Dinâmica*. Esta explanação não responde o ítem **(i)**; entretanto, caracteriza os requisitos básicos para respondê-lo, visto que um trecho de código qualquer (recorrente ou não) é redundante se composto por instruções redundantes.

Considerando **(i)** como premissa, pode-se verificar **(ii)** a partir das explanações e procedimentos expostos nos parágrafos a seguir:

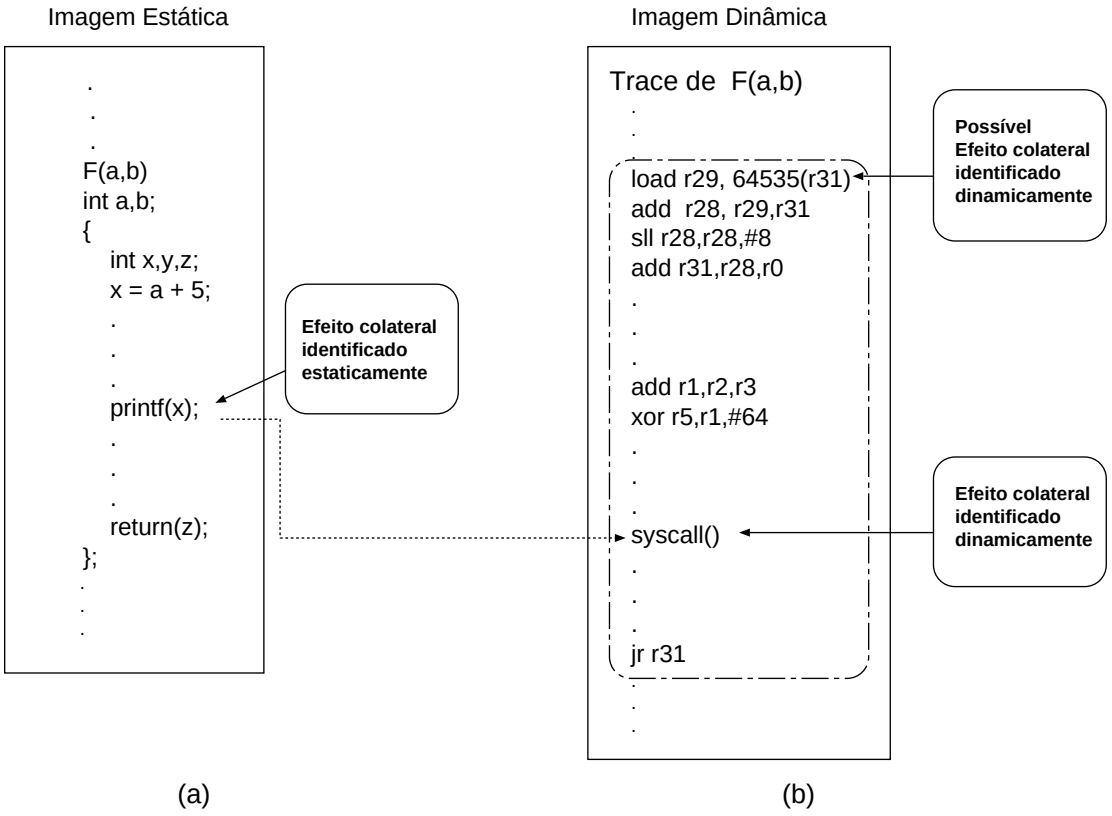


Figura 1.4: Impedimentos à exploração de redundância, (a) representação estática, (b) representação dinâmica.

Na Figura 1.4(a) é esboçada a representação estática codificada em linguagem de alto nível de uma função **F(a,b)**, com parâmetros de entrada limitado aos valores instanciados à **a** e **b** (para efeitos de exemplificação, os valores instanciados aos parâmetros de entrada repetem-se freqüentemente, o que torna a função redundante). Neste exemplo, a função **F(a,b)** não permite a exploração de reuso, pois existe uma instrução (*printf(x)*) interna à **F(a,b)**, que precisa ser executada para que os resultados da computação apresentem-se corretos. Instruções com esta característica,

serão classificadas como *instruções com efeitos colaterais*, pois produzem incorreções na execução de um programa em decorrência de serem reusadas internamente a um trecho de código redundante. São exemplos destas instruções: operações sobre variáveis globais, ponteiros, funções do sistema, e outras (dependendo do contexto de análise).

A Figura 1.4(b), apresenta a representação dinâmica da função **F(a,b)** (esta também será redundante, pois é imagem dinâmica da função estática **F(a,b)**). Observa-se, neste caso, que possíveis tentativas de memorização dinâmica são negativamente agravadas pelo aumento da possibilidade de ocorrência de instruções que produzem efeitos colaterais. Entre outras considerações, pode-se observar que a instrução **load r29,64535(r31)** pode apresentar efeitos colaterais. Mesmo apresentando um mesmo valor de entrada para o registrador **r31** e logo o mesmo endereço de memória calculado, o valor armazenado neste endereço de memória pode variar. Para este caso, o reuso produzirá valores inesperados, que podem implicar na incorreção dos resultados produzidos pela função ou influenciar negativamente em outras fases da execução. Este problema pode ser solucionado, caso existam mecanismos que indiquem dinamicamente as inconsistências de memória impedidas de sofrerem operações de reuso.

Uma tentativa para amenizar as restrições impostas pelas instruções com efeitos colaterais, seria evitar o reuso destas, particionando-se a função (ou trecho de código) de modo a excluí-las da aplicação do reuso. A Figura 1.5(a), apresenta uma possível solução do problema para o caso estático. Para o caso dinâmico na Figura 1.5(b), o trace considerado (da função), também deverá ser particionado de modo a evitar instruções com efeitos colaterais. Pode-se observar agora que, duas novas funções, **F1(a,b)** e **F2(a,b)**, podem agora explorar parcialmente a redundância da função **F(a,b)**. A mesma consideração pode ser aplicada ao caso dinâmico para os traces **Trace1** e **Trace2**. Embora a solução para o caso estático pareça ser viável, esta não é genérica e é aplicada somente para casos particulares, onde as instruções com efeito colaterais são visíveis. Identificar estaticamente instruções com efeito colaterais inibitórias ao reuso, sendo pela inspeção do código ou por auxílio do compilador, representa, ainda (dado o estado da arte), uma tarefa extremamente complexa [42, 28, 12]. Contrariamente ao caso estático, no caso dinâmico, tais restrições

podem ser contornadas com um grau de complexidade bem inferior. Contudo, a solução para o caso dinâmico produz uma fragmentação considerável, decorrente do particionamento necessário para evitar o reuso de instruções com efeitos colaterais nos trechos de código a serem identificados como redundantes e conseqüentemente reusados.

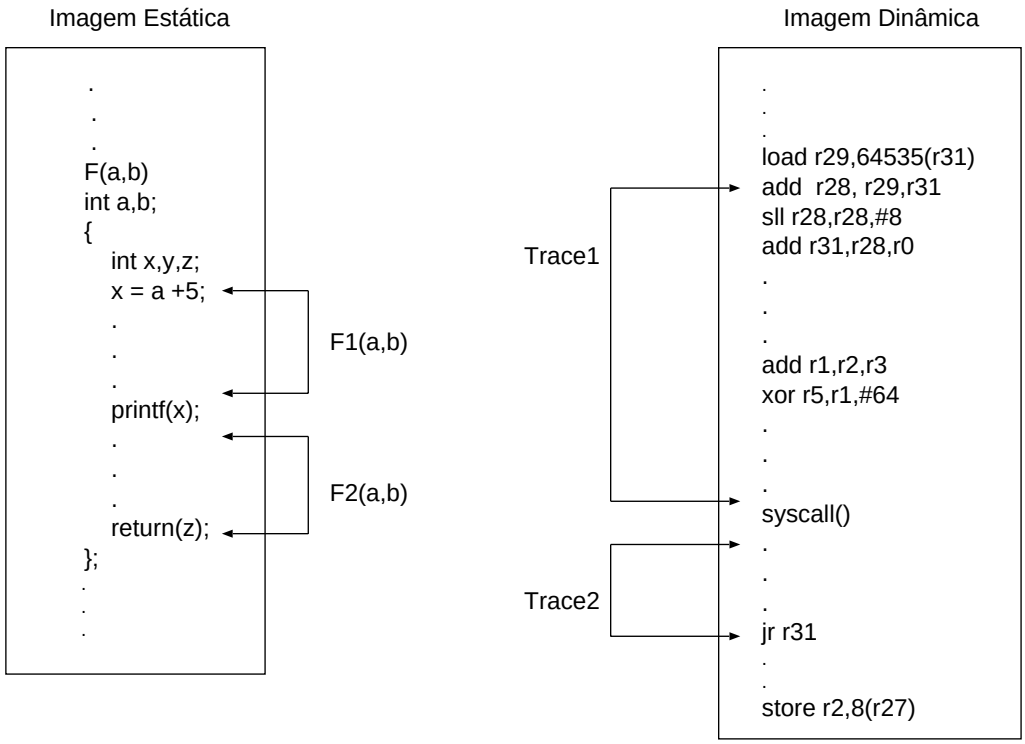


Figura 1.5: Amenizando restrições impostas por instruções com efeitos colaterais, (a) solução estática, (b) solução dinâmica.

A partir do exposto nos parágrafos anteriores, não é possível responder **(iii)** de modo integral e completamente ideal. Entretanto, aproximações podem levar a soluções satisfatórias. Uma das aproximações que será avaliada neste trabalho de tese consiste em estabelecer um mecanismo em nível de arquitetura de processador, com o objetivo de encapsular dinamicamente o maior trecho de código possível, possuindo este, comportamento repetitivo observado durante sua execução e livre de efeitos colaterais.

1.4 Objetivos

O objetivo principal deste trabalho é investigar e propor uma alternativa para explorar o reuso de computações redundantes, de forma dinâmica e em nível de arquitetura de processador. Para alcançar este objetivo, é proposto um mecanismo objetivando explorar o reuso de traces (seqüências de instruções dinâmicas), e a incorporação deste mecanismo em uma arquitetura de um processador superescalar. Caberá ao mecanismo proposto: a identificação dos traces redundantes; o tabelamento dos traces candidatos a reuso; e a identificação de oportunidades de reuso.

A seqüência de investigações efetuadas para verificar a viabilidade de se explorar reuso em nível de traces, serão:

(i) Verificar o volume de instruções dinâmicas sem efeitos colaterais e que são redundantes durante a execução de um programa;

(ii) Verificar a partir das instruções selecionadas em (i), se existem agrupamentos seqüenciais de instruções que determinam caminhos de execução (traces);

(iii) Determinar heurísticas que permitam capturar os traces determinados em (ii);

(iv) Avaliar o potencial de redundância de um mecanismo implementado pela heurística determinada em (iii);

(v) Avaliar o ganho de performance do mecanismo proposto em (iv), incorporando-o a um processador superescalar;

(vi) Investigar as características e requisitos dos traces obtidos em (iii) e do processador proposto em (v);

(vii) Propor novas alternativas e perspectivas.

1.5 Contribuições

Este trabalho de tese apresenta como contribuições principais:

• A introdução do conceito de memorização dinâmica de traces e definição de um mecanismo capaz de identificar construtivamente, memorizar e reusar traces, denominado *Dynamic Trace Memoization (DTM)*;

• Comprovação da existência de redundância com granularidade em nível de traces, e medidas que discriminam a quantidade de reuso identificada, composição e características dos traces redundantes;

• Proposta de uma heurística para identificação construtiva de traces redundantes. A heurística proposta é componente do mecanismo de reuso e determina as regiões de análise, independente de seus limites naturais e intuitivos (instruções e blocos básicos);

• Avaliações de performance de uma arquitetura superescalar incorporando o mecanismo *DTM*, das restrições impostas pelo modelo de execução superescalar à exploração de reuso, dos requisitos exigidos pelo mecanismo proposto e como estes interagem com a arquitetura superescalar substrato.

1.6 Estrutura do Trabalho

Este texto está dividido em 6 partes. A primeira parte, introdutória, inclui o presente capítulo. No Capítulo 2 é apresentado o estado da arte em exploração de computações redundantes em nível de arquitetura de processador, suas diferentes granularidades e mecanismos propostos. A terceira parte é composta pelo Capítulo 3, onde é apresentada uma solução alternativa aos mecanismos descritos no Capítulo 2. Neste, é apresentado o mecanismo *DTM* e a descrição de seu funcionamento. No Capítulo 4, o mecanismo *DTM* é incorporado à uma arquitetura superescalar substrato e são discutidos os detalhes de sua adaptação. No Capítulo 5 é apresentado o ambiente experimental e os resultados dos experimentos efetuados. No Capítulo 6 são apresentadas as conclusões e direcionamentos futuros desta pesquisa.

Capítulo 2

Trabalhos Relacionados

Neste capítulo serão apresentados os trabalhos diretamente relacionados a exploração de computações redundantes em nível de arquitetura de processador.

2.1 O Value Cache

Em [32], é proposta uma arquitetura de processador com o objetivo de atenuar a necessidade e o custo de construção de compiladores complexos e orientados à otimização do código objeto gerado a partir de linguagens de alto nível [2, 9]. Para alcançar este objetivo, foi incluído na arquitetura um mecanismo denominado *Value Cache*. O *Value Cache* é responsável pela tarefa de identificação e eliminação de expressões redundantes que ocorrem em tempo de execução. Classifica-se este mecanismo como *ID-D/ME-H/OP-H*.

As expressões redundantes tratadas pelo mecanismo são restritas a expressões simples que não possuam: instruções de desvio, chamadas de procedimentos e escrita em memória. O contexto de entrada (*dependency set*), os valores instanciados ao contexto e o resultado da expressão a ser tratada, são obtidos durante a execução da mesma, e armazenados em uma entrada de uma memória interna ao mecanismo (*Value Cache*), sendo este embutido no processador. Se uma mesma expressão for executada em uma computação posterior e o contexto de entrada da expressão não tiver sido reescrito por nenhuma outra instrução neste período de tempo (desde o cacheamento da expressão até uma próxima reexecução da mesma expressão), então o resultado já armazenado será reusado, evitando-se assim a reexecução da

expressão. Caso contrário, a expressão armazenada será marcada como inválida, não sendo portanto reusada.

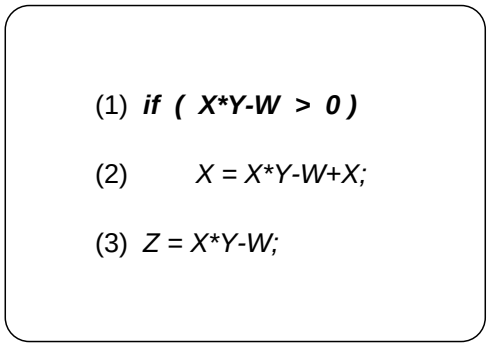


Figura 2.1: Trecho de código possuindo uma expressão redundante que só pode ser eliminada dinamicamente.

Considerando a Figura 2.1, identifica-se neste exemplo uma seqüência de código em alto nível possuindo a expressão simples $X * Y - W$ como expressão redundante (onde X, Y e W formam o contexto de entrada da expressão). Supondo que a expressão $X*Y-W$ em (1) foi executada (pois não se encontra armazenada no *Value Cache*), então esta expressão será armazenada em uma entrada de *Value Cache*, conjuntamente com o seu contexto de entrada (X, Y e W), os valores instanciados aos operandos do contexto de entrada, e o resultado decorrente da execução da expressão.

Dando seqüência a execução do trecho de código apresentado:

• Se em (1) for produzido um valor verdadeiro, então a execução de (2) irá explorar a repetição da expressão $X*Y-W$, já que esta foi executada em (1) e o contexto de entrada da expressão não foi reescrito por nenhuma instrução intermediária. Neste contexto, a instrução em (3) não poderá reusar o resultado da expressão, pois (2) escreve em um dos elementos do contexto de entrada da expressão (no caso, o valor da variável X), invalidando assim o seu reuso.

• Se em (1) for produzido um valor falso, então (2) não será executada e não reescrevendo portanto no contexto de entrada da expressão, permitindo assim o reuso da expressão em (3).

O exemplo apresentado, caracteriza a impossibilidade do compilador em eliminar

expressões redundantes efetuando apenas a análise estática do código. Pode-se observar que o método proposto não explora a repetição dos valores de entrada. Este procura somente eliminar expressões redundantes e não explorar o reuso de trechos de código ou de expressões genéricas.

2.2 O Result Cache

Em [53], é feito um estudo experimental sobre a comportamento redundante (repetitivo) de trechos de código, procedimentos, funções e instruções com alta latência de execução, que são encontrados em programas. Verificou-se neste trabalho que para os programas avaliados, determinadas regiões de código operam sobre dados de entrada que repetem-se freqüentemente. A partir da observação deste comportamento, foi proposta a aplicação da técnica de memorização (ou tabulação de resultados). Esta, foi aplicada a funções, procedimentos ou regiões de código (não necessariamente recorrentes) presentes em programas e previamente selecionadas pelo programador (*ID-E/ME-S/OP-S*).

O método é descrito pelo seguinte algorítmo:

1. Uma vez ativada uma função, seus parâmetros de entrada são instanciados direta ou indiretamente à valores. Sobre a função instanciada agora pelos seus parâmetros de entrada, é aplicada uma função com o objetivo de formar uma chave de busca para acesso à tabela de memorização (*Result Cache*);

2. Uma vez efetuada uma pesquisa na tabela

   caso ocorra sucesso :

   a função não será reexecutada, sendo o seu resultado (já armazenado em *Result Cache*) fornecido pela correspondente entrada na tabela;

   caso contrário (em não se encontrando a entrada correspondente):

   a função instanciada será então normalmente executada e ao produzir um resultado, este será armazenado em uma entrada da tabela de memorização.

O método exposto avalia a implementação da tabela de memorização em nível de software. Para a aplicação de memorização, as seguintes condições devem ser

satisfeitas:

• As regiões a serem memorizadas não devem possuir instruções com efeitos colaterais;

• É exigida a intervenção do programador para a delimitação dos trechos de código a serem memorizados.

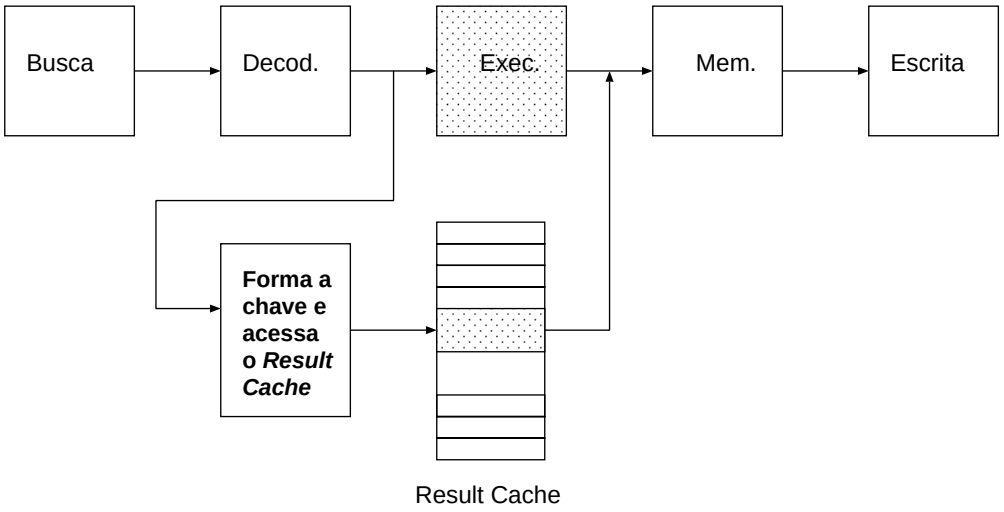


Figura 2.2: O *Result Cache* implementado em nível arquitetural.

Uma segunda proposta apresentada no trabalho, propõe a aplicação de memorização dinâmica *ID-D/ME-H/OP-H* para instruções com alta latência de execução (multiplicação, divisão e raiz quadrada). Nesta proposta, instruções com alta latência de execução são candidatas automáticas para reuso, e são cacheadas em uma memória interna ao processador (referenciada como *Result Cache*). Na Figura 2.2, é apresentada uma possível implementação do mecanismo. Quando uma instrução com alta latência de execução é decodificada e seus operandos lidos (dados de entrada), estes são utilizados para formar uma chave que indexa o *Result Cache*. Após a formação da chave de busca o *Result Cache* é acessado. Caso seja encontrada uma entrada que confirme a redundância da instrução pesquisada, o valor armazenado na respectiva entrada será reusado e a execução em curso (na respectiva unidade funcional) será abortada. Para o caso em que não seja identificada uma entrada no *Result Cache*, a instrução corrente será executada e o resultado produzido será armazenado em alguma entrada do *Result Cache*.

2.3 O Cache de Resultados

Com o objetivo de acelerar o processamento de programas, foi proposto e avaliado em [52], um mecanismo de memorização *ID-E/ME-H/OP-H*. O mecanismo desenvolvido é aplicado à funções codificadas em alto nível, onde tais funções são pré-selecionadas pelo programador. Este mecanismo utiliza uma tabela interna ao processador, denominada *Cache de Resultados*, e que possui a função de armazenar os contextos de entrada e saída (resultados) de execuções prévias das funções selecionadas. O mecanismo proposto incorpora instruções especializadas ao processador e que são inseridas no código objeto do programa alvo. Estas são inseridas de modo a contornar as instruções que ativam a execução das funções a serem memorizadas, e que quando executadas efetuam uma busca na tabela de memorização utilizando como chave um valor numérico associado a função e o respectivo contexto de entrada. Quando da ocorrência de uma busca na tabela de memorização, caso seja encontrada no *Cache de Resultados* uma entrada que coincida com a função instanciada, então os resultados armazenados nesta entrada são atribuídos ao contexto de saída da função (resultado), evitando-se deste modo a reexecução da função; caso contrário, a função em questão será executada normalmente e a execução de uma instrução especializada incluirá uma nova instância da função e seus resultados no *Cache de Resultados*.

A Figura 2.3(a), apresenta uma seqüência de código em alto nível que faz uma chamada a uma função **Fat(a)**, sendo esta, instanciada com seu respectivo argumento de entrada. Na Figura 2.3(b) é apresentada a mesma seqüência de código, porém em código objeto. Na Figura 2.3(c) é apresentada a mesma seqüência de código incluindo instruções especializadas para a efetuação de memorização. As instruções especializadas são executadas com o objetivo de verificar se a função **Fat(a)** apresenta alguma execução prévia já cacheada no *Cache de Resultados*. Se existir, o resultado será recuperado e a função não será executada. Caso contrário, a função **Fat(a)** será executada e as instruções especializadas incluirão a função e seu contexto de entrada/saída no *Cache de Resultados*.

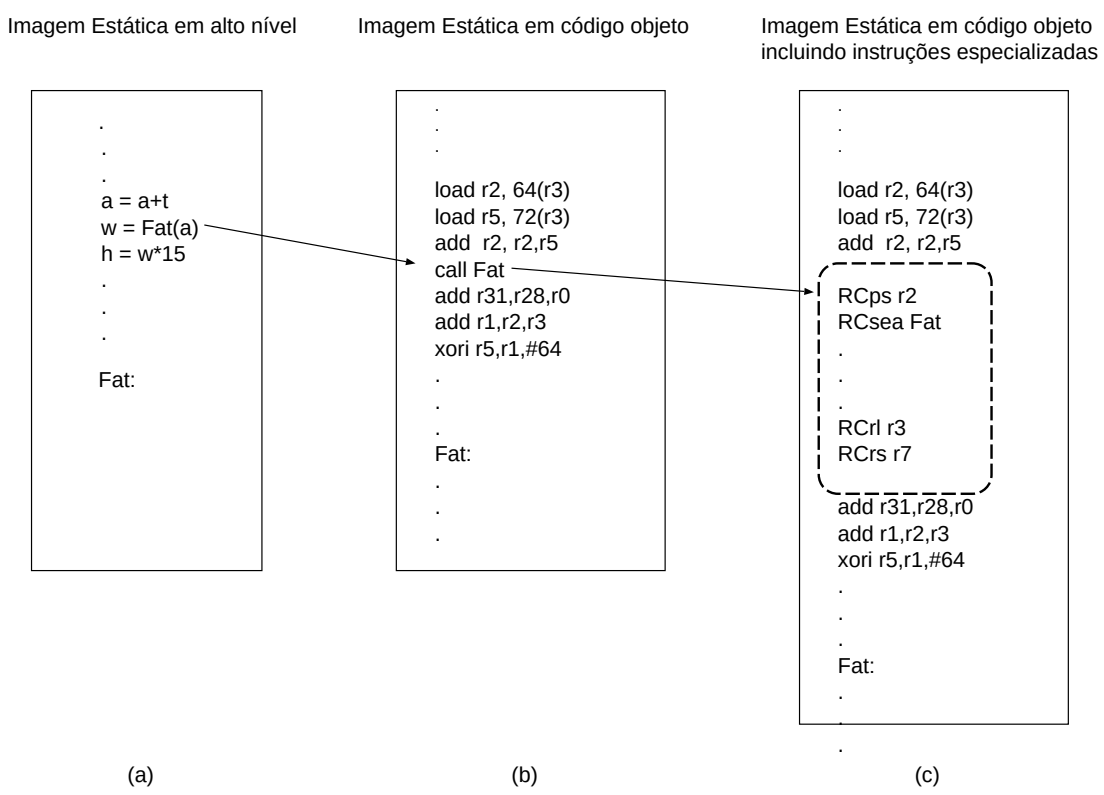


Figura 2.3: Delimitando uma função a ser memorizada, (a) código em alto nível, (b) código objeto, (c) código objeto incluindo instruções especializadas para ativar memorização.

2.4 Load Value Prediction, Value Prediction e a predição de valores

Embora não seja uma técnica que explore computação redundante através de reuso, a predição de valores utiliza-se da memorização. A predição de valores está intimamente relacionada com computação redundante e obtém proveitos desta através de especulação. Em [45] é apresentado um mecanismo denominado *Load Value Predition* (*LVP*), que explora a localidade de valores armazenados em memória. Neste trabalho, observou-se empiricamente que durante a execução de programas do *SPEC92* e *SPEC95*, um grande percentual de instruções de acesso à memória (loads) produzem valores já observados anteriormente. Baseado neste comportamento repetitivo, foi proposto um mecanismo para explorar tal fenômeno, de modo a permitir a predição do valor a ser produzido por uma instrução de acesso à memória.

Em [44], a aplicação do conceito de localidade de valores é estendida para pro-

ver através de predição, os valores a serem instanciados à quaisquer registradores internos do processador (não restringindo a aplicação de predição à registradores alvo de instruções de acesso à memória). Nesta nova aproximação, o mecanismo responsável pela predição foi rebatizado como *Value Prediction* (*VP*), entretanto o funcionamento básico permanece inalterado. O *VP* efetua dinamicamente o tabelamento (em um cache interno ao processador) dos n últimos valores instanciados individualmente aos registradores de destino (resultado) de cada instrução dinâmica observada. Quando instruções previamente observadas forem referenciadas, o valor do registrador que armazenará o resultado de sua execução será predito. Basicamente, a predição de valores *explora computação redundante de forma especulativa*, e para tanto, necessita de um mecanismo com a função de validar os valores preditos, afim de manter a correção da execução dos programas. É conveniente ressaltar que em caso de uma predição incorreta, paga-se uma penalidade correspondente ao tempo necessário para se restaurar o contexto do processador para o estado anterior à predição efetuada.

É importante observar que estes trabalhos exploram computações redundantes com granularidade em nível de instruções e de forma especulativa. Torna-se evidente que *a predição de valores* é vantajosa apenas, se esta puder ser feita com precisão satisfatória, desde que predições incorretas ocasionam além das penalidades mencionadas, um aumento dos conflitos estruturais. Simulações do mecanismo abordado, identificaram que quando uma alta taxa de acerto sobre as predições efetuadas é alcançada, o mecanismo promove a redução de caminhos críticos provocados por dependências verdadeiras, reduzindo assim alguns dos impedimentos impostos para a obtenção de um maior grau de paralelismo.

A Figura 2.4, apresenta um exemplo de um mecanismo para efetuar a predição de valores. Supondo que a instrução de leitura à memória no endereço *100* sofrerá uma predição, o endereço da instrução de memória é utilizado para acessar uma tabela que armazena em cada uma de suas entradas, os **n** últimos valores instanciados ao registrador de destino da operação de acesso à memória (*r1* para o caso analisado). Caso ocorra uma *falha*, ou seja, o endereço pesquisado não está presente na tabela, então nenhuma predição será efetuada. Caso ocorra *sucesso* na busca, a entrada selecionada irá fornecer ao preditor, até **n** valores candidatos à predição. O preditor

irá avaliar e selecionar dentre os valores considerados, o que possui a maior probabilidade de ser o correto. Dado que um valor foi escolhido, este será instanciado ao registrador *r1* e as instruções subseqüentes (que dependem do valor de *r1*) serão despachadas para execução especulativa enquanto o valor exato a ser produzido pela instrução *100* não se tornar disponível. Quando o valor real do registrador *r1* for instanciado a este, este valor será comparado ao valor anteriormente predito. Se a comparação anterior validar a igualdade dos valores, então as instruções marcadas como especulativas serão desmarcadas como especulativas e finalizadas normalmente; senão, as instruções especulativas serão descartadas e despachadas com o valor real. Para ambos os casos o preditor atualizará as informações de predição.

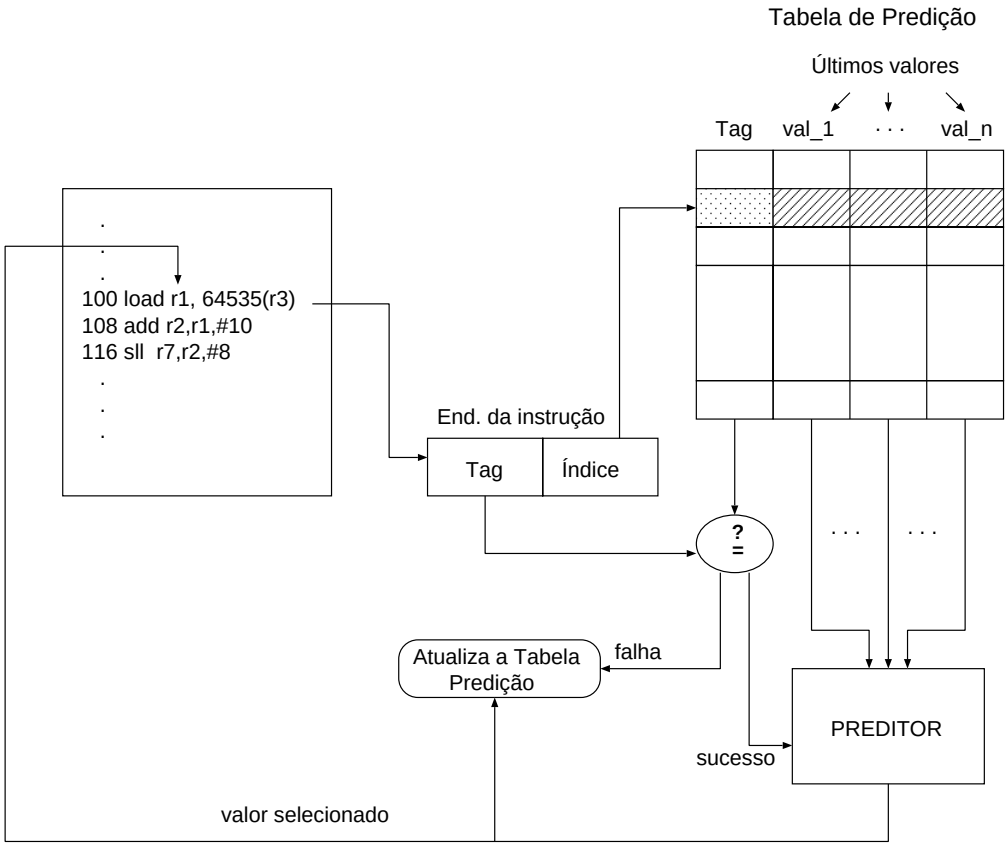


Figura 2.4: Mecanismo básico para predição de valores.

Em [29], são propostos e avaliados alguns mecanismos de predição de valores baseados no comportamento redundante de instruções (localidade temporal de valores) e no contexto de valores gerados pelas instruções.

2.5 O Reuso de Instruções Dinâmicas

Em [63] é introduzido o conceito de reuso de instruções dinâmicas. Neste trabalho, explora-se computação redundante com granularidade em nível de instruções. O conceito é avaliado considerando a sua incorporação em uma arquitetura superescalar. São propostos três esquemas projetados para explorar o reuso de instruções dinâmicas. Todos os esquemas consideram uma memória interna ao processador e denominada *Reuse Buffer* (*RB*). O *RB* armazena em tempo de execução: os endereços de memória das instruções instanciadas; seus operandos de entrada e o valor resultante da execução da instrução. O objetivo dos esquemas propostos, é evitar a reexecução de instruções, verificando em tempo de execução se estas estão presentes no *RB*. Os três esquemas consideram que somente as instruções mais recentemente executadas serão inseridas no *RB*. É assumido também, que instruções só poderão ser pesquisadas no *RB*, se estas estiverem instanciadas (possuírem valores associados a todos os seus operandos de entrada). Todos os esquemas de reuso apresentados, classificam-se como *ID-D/ME-H/OP-H*.

Na Figura 2.5, são apresentadas as diferentes composições das entradas do *RB* para cada um dos esquemas propostos.

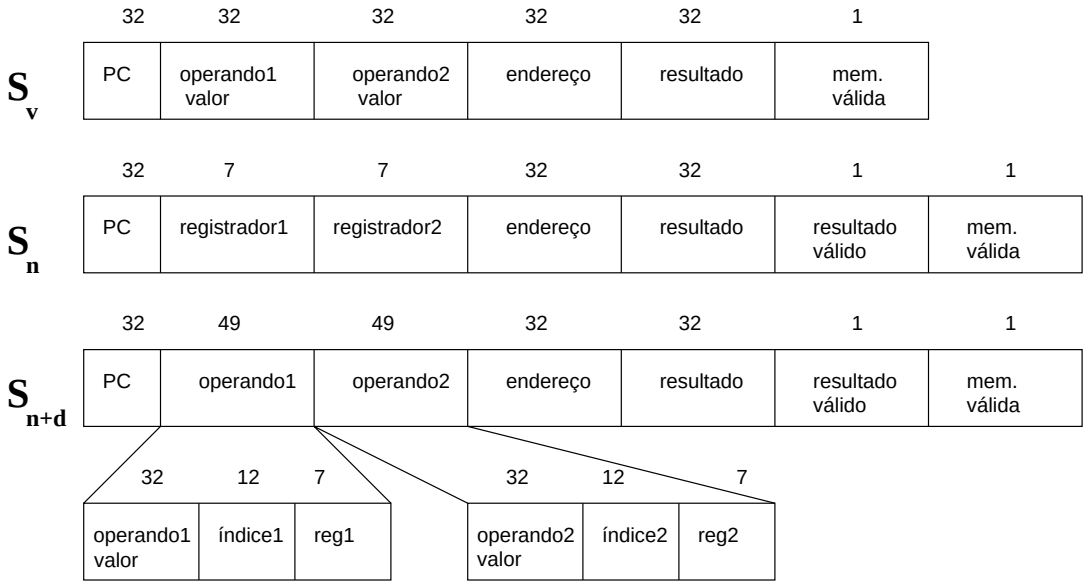


Figura 2.5: Composição das entradas do *RB* para os esquemas de reuso S_v, S_n, S_{n+d}.

As subseções seguintes irão explanar cada um dos esquemas de reuso considerados

em [63].

2.5.1 O esquema S_v

O esquema S_v explora o reuso através da identificação de repetição dos valores de entrada. A seguir serão discriminados os elementos de uma entrada do *RB* para suportar o esquema S_v.

PC - Armazena o endereço de memória da instrução cacheada;

operando1(2) valor- Armazenam os valores dos operandos de entrada;

endereço - Armazena o endereço de acesso de uma operação de memória;

resultado - Armazena o valor resultante da execução da instrução;

mem. válida- Indica a validade do reuso de inst. de memória.

Descrição do esquema S_v:

Quando uma instrução no estágio de decodificação é instanciada com seus operandos de entrada (se estes estiverem prontos), a instrução em questão será pesquisada em *RB*. Se encontrada, então será reusada, senão será executada normalmente e posteriormente inserida em *RB*. As instruções de leitura de memória (Ex: *load r1,0(r3)*), possuem tratamento diferenciado, pois podem produzir efeitos colaterais. Um *load* instanciado e presente em *RB* nem sempre poderá ser reusado, pois o conteúdo de memória referente a um endereço reusado pode não ser redundante. Um *load* somente será reusado se os operandos de entrada forem repetidos (portanto reusado o endereço calculado) e se o conteúdo de memória referenciado pelo endereço repetido não tiver sido escrito desde o instante em que o *load* foi inserido no *RB*. Para verificar esta condição, a cada escrita efetuada na memória, o *RB* deverá ser atualizado de forma a invalidar (não permitir reuso) as entradas que fazem referência ao endereço de memória onde ocorreu a escrita (este procedimento é implementado em todos os esquemas). É importante notar que a permamente validação de instruções de acesso à memória no *RB* provoca uma busca associativa no mesmo, via o campo *endereço*.

A Figura 2.6 ilustra o reuso efetuado pelo esquema S_v. Para este exemplo, a instrução no endereço *10C* será reusada se estiver presente em *RB* e se os valores de seus operandos de entrada *r4* e *r7* presentes na entrada selecionada, forem iguais

aos seus respectivos valores de *r4* e *r7* no arquivo de registradores do processador. Para o caso em que as condições expostas acima sejam satisfeitas, a instrução será reusada (não será executada e o resultado será armazenado no registrador *r3*). Caso contrário, a instrução será executada e incluída com seus operandos de entrada, saída, e os respectivos valores associados.

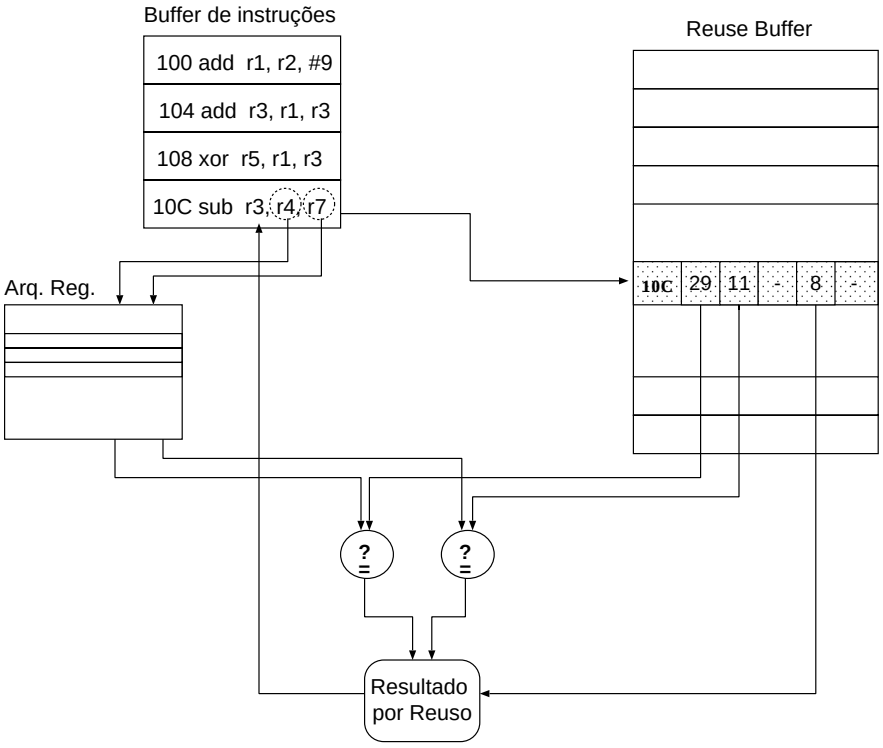


Figura 2.6: Acesso ao *Reuse Buffer* e identificação de instruções redundantes pelo esquema S_v.

2.5.2 O esquema S_n

O esquema S_n explora o reuso, avaliando o estado de seus operandos de entrada. A seguir serão discriminados os elementos de uma entrada do *RB*, para suportar o esquema S_n.

PC - Armazena o endereço de memória da instrução cacheada;

registrador1(2) - Armazenam os nomes dos operandos de entrada;

endereço - Armazena o endereço de acesso de uma operação de memória;

resultado - Armazena o valor resultante da execução da instrução;

resultado válido - Indica se o resultado armazenado no campo *resultado* é válido;

mem. válida - Indica a validade do reuso de inst. de memória.

Descrição do esquema S_n:

O esquema S_n procura simplificar o teste de reuso. Os identificadores dos operandos de entrada de cada instrução (registradores) são armazenados em RB ao invés dos valores instanciados aos operandos. Quando uma instrução escreve em um registrador, são invalidadas todas as instruções presentes em RB que possuam tal registrador como operando fonte. Dado que uma instrução i foi incluída em RB, i será posteriormente reusada, se nenhuma outra instrução subseqüente a i escreveu em quaisquer um dos operandos de entrada de i, ou seja, se i armazenada em RB não foi invalidada. Este esquema de reuso reedita o esquema apresentado em [32].

A Figura 2.7 apresenta um exemplo de funcionamento do esquema S_n. Para a seqüência dinâmica apresentada, somente as instruções *100* e *104* podem ser reusadas, pois seus operandos de entrada não são reescritos por nenhuma instrução interna ao laço. Para este caso, a permissão de reuso é dada pelo campo válido. As demais instruções (*108*,*10C* e *110*) não podem ser reusadas, pois seus operandos são reescritos, invalidando assim o valor do campo resultado. Apesar da seqüência de código poder ser questionada quanto a sua possível otimização estática (mover as instruções *100* e*104* para fora do laço), este procedimento só poderá ser garantido, se for determinado que nenhuma outra instrução do laço é alcançável por um desvio a partir de qualquer outra instrução dinâmica.

2.5.3 O esquema S_{n+d}

O esquema S_{n+d} explora o reuso, avaliando o estado dos operandos de entrada e a relação de dependência entre instruções. A seguir serão discriminados os elementos de uma entrada do *RB* para suportar o esquema S_{n+d}.

PC - Armazena o endereço de memória da instrução cacheada;

operando1 e operando2 - São compostos pelos seguintes sub-campos:

operando valor - Armazena o valor do operando;

índice - Armazena o endereço de encadeamento na tabela;

reg - Armazena a referência ao registrador usado como operando.

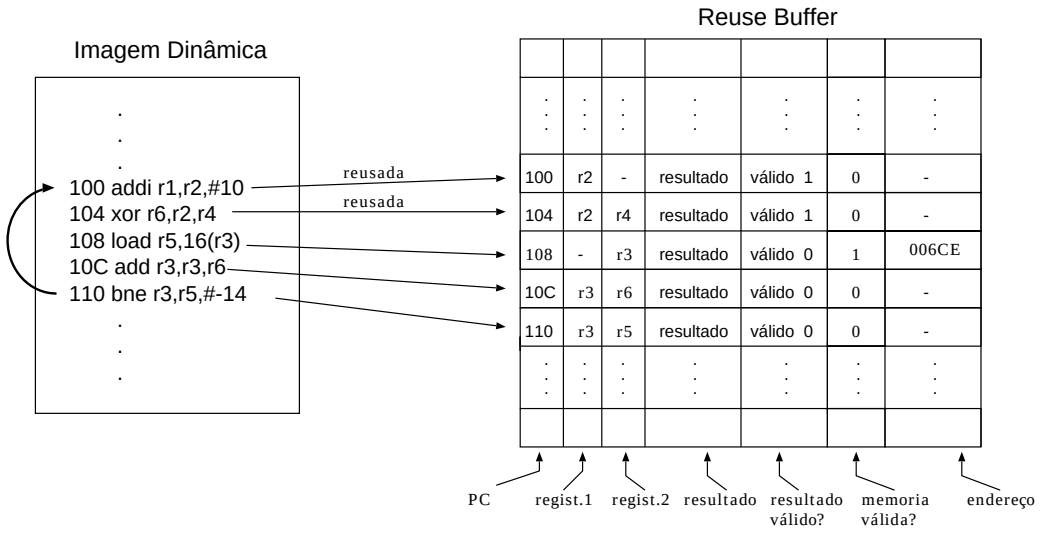


Figura 2.7: Exemplo de reuso adotado pelo esquema S_n.

endereço - Armazena o endereço de acesso de uma operação de memória;

resultado - Armazena o valor do resultado da execução da instrução;

resultado válido - Indica se o resultado armazenado no campo *resultado* é válido;

mem. válida - Indica a validade do reuso de inst. de memória.

Descrição : O esquema S_{n+d} estende o esquema S_n para tratamento de cadeias de instruções dependentes (seqüências de instruções que possuem entre si dependências de dados verdadeiras). A presente descrição estende o esquema S_{n+d} como apresentado em [65]. Cadeia de instruções dependentes são inseridas no RB, armazenando-se em cada entrada, informações sobre a instrução componente da cadeia (de acordo com os campos da entrada de RB descrita anteriormente). Estas são ligadas via o campo *índice*, na ordem das instruções que produzem valores para as que consomem os valores produzidos. A Figura 2.8 apresenta um grupo de instruções aptas para o despacho (supondo os operandos de entrada prontos). O RB será acessado pelos endereços das instruções e posteriormente pelas instruções instanciadas (os acessos são efetuados durante as fases de busca e decodificação). Quando estas forem identificadas como presentes no RB, torna-se ativa uma avaliação das relações *produtor/consumidor* entre as instruções identificadas, varrendo-se para isso os campos de encadeamentos. No exemplo exposto, as instruções apresentam todos os seus operandos dependentes da instrução *100*, logo, uma vez que esta seja redundante,

conseqüentemente todas as outras também serão redundantes e portanto reusadas.

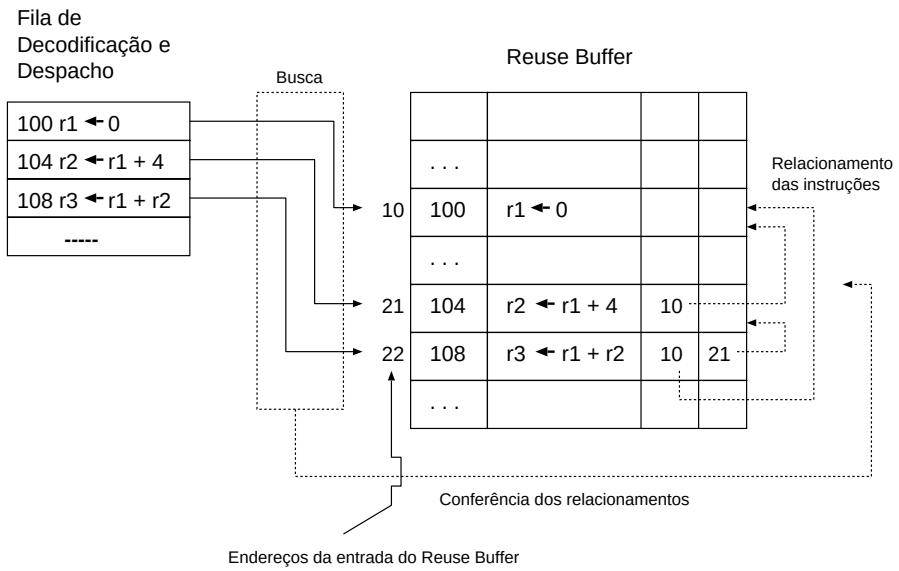


Figura 2.8: Acesso ao *RB* e identificação de cadeias redundantes aptas para o reuso.

2.6 O Block History Buffer

Em [35] é apresentada a exploração de reuso sobre computações redundantes com granularidade em nível de blocos básicos. Neste trabalho é proposto um mecanismo denominado *BHB- block history buffer*. O mecanismo proposto é baseado no tabelamento dinâmico dos contextos de entrada/saída de blocos básicos, em uma estrutura implementada em hardware (*BHB*). O mecanismo apresenta-se como uma alternativa ao reuso aplicado em nível de instruções dinâmicas. Este mecanismo pode ser classificado como *ID-E/ME-H/OP-S*, para o caso em que os blocos básicos são marcados durante a compilação, e *ID-D/ME-H/OP-H* quando estes são determinados dinamicamente.

Blocos básicos podem der identificados dinamicamente da seguinte forma:

(i) Qualquer instrução após um desvio é identificada como ponto de entrada de um novo bloco básico. A primeira instrução de um programa é automaticamente um ponto de entrada de um bloco básico;

(ii) Uma instrução de desvio marca o final de um bloco básico. Chamadas e

retornos de subrotinas são tratados exatamente como qualquer outra instrução de desvio;

(iii) Um desvio para alguma instrução interna a um bloco básico, divide o bloco básico corrente em dois blocos básicos.

A Figura 2.9, apresenta a composição de uma entrada da *BHB*, a descrição de cada um dos campos especificados é a seguinte:

PC-Block : Endereço da primeira instrução do bloco básico;

Reg-In :

Reg- Registrador participante do contexto de entrada;

Data- Valor instanciado ao registrador *Reg*;

Reg-Out :

Reg- Registrador participante do contexto de saída;

Data- Valor instanciado ao registrador *Reg*;

Mem-In :

PC- Endereço da instrução de acesso à memória (leitura);

Addr- Endereço de memória a ser acessado;

Data- Valor armazenado no endereço *Addr*;

Full- Indica se a entrada está ativa;

Mem-Out :

PC- Endereço da instrução de acesso à memória (escrita);

Addr- Endereço de memória a ser acessado;

Data- Valor armazenado no endereço *Addr*;

Full- Indica se a entrada está ativa;

Next-Block : Endereço da próxima instrução subseqüente ao bloco básico.

Basicamente o mecanismo funciona da seguinte forma:

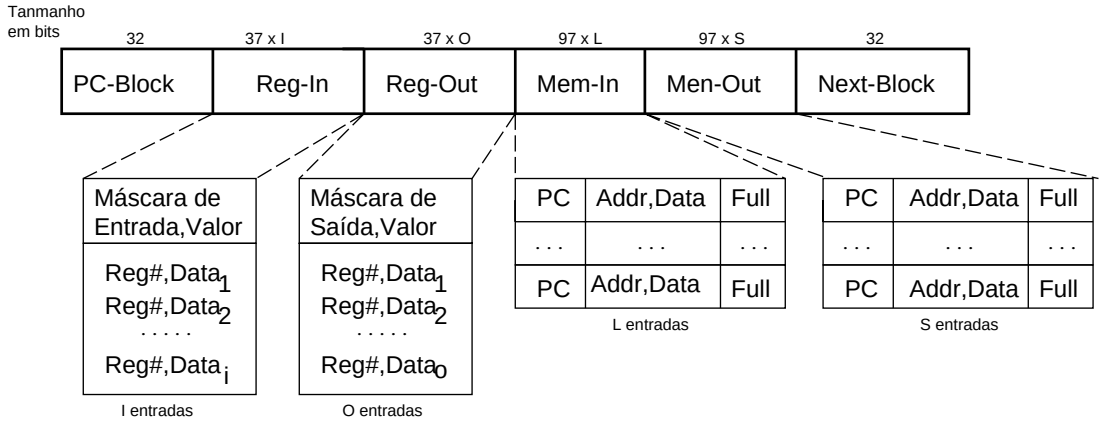


Figura 2.9: Composição de uma entrada do *BHB*.

(i) Os limites dos blocos básicos são marcados durante a compilação, ou para alguns casos, determinados em tempo de execução (embora o contexto de entrada/saída seja determinado e marcado, em tempo de compilação);

(ii) Quando um bloco básico é executado, seus contextos de entrada e saída são armazenados em *BHB*;

(iii) Antes da execução de um bloco básico, a *BHB* é pesquisada para identificar se o bloco básico corrente já foi executado com os mesmos valores atuais do contexto de entrada. Em caso afirmativo, o bloco básico é reusado, ou seja, as instruções pertencentes ao bloco básico não são **reexecutadas** e o contexto de saída é atualizado com as informações presentes em *BHB*.

Descrição:

Quando uma instrução é lida da memória (estágio de busca), a *BHB* é pesquisada. Se o endereço da instrução lida combina com alguma entrada de *BHB* (com o endereço da primeira instrução de um bloco básico armazenado em *BHB*), então para a entrada selecionada, os valores referenciados pelo contexto de entrada são comparados com os respectivos valores do arquivo de registradores e de memória. Se os valores comparados forem iguais, então todas as instruções do bloco básico serão reusadas, ou seja, não serão reexecutadas. Para este caso, os valores referenciados pelo contexto de saída (e armazenados na mesma entrada de *BHB*), serão escritos nos respectivos operandos de saída do bloco básico. Instruções de acesso à memória (leitura) que integram um bloco básico, deverão ser executadas antes

da comparação, pois estas possuem efeitos colaterais (mesmo possuindo os mesmos operandos de entrada podem prover resultados diferentes). Instruções de escrita em memória inclusas no bloco básico reusado, deverão ser executadas de forma a garantir a consistência de memória.

A Figura 2.10 apresenta um exemplo de reuso de bloco básico (supondo o bloco básico em questão já armazenado em *BHB*). Quando a instrução no endereço de memória *100* for acessada, simultaneamente, será efetuada uma busca em *BHB* para identificar uma entrada com *PC-Block = 100*. Dado que uma entrada de *BHB* foi selecionada para avaliação, o contexto de entrada parcial (referenciados por *r7* e *r9*, pois estes determinam o contexto de entrada das instruções de leitura à memória), será comparado aos respectivos valores no arquivo de registradores. Se a comparação resultar em sucesso (acusando desta forma uma redundância parcial do bloco básico), então as instruções de memória inclusas no campo *Mem-In* e marcadas como ativas pelos subcampos *Memo-In/Full*, acessarão o cache de dados a partir dos endereços armazenados nos subcampos *Memo-In/Addr*. Se os valores lidos do cache coincidirem com os valores armazenados nos respectivos subcampos *Memo-In/Data*, então a redundância total foi alcançada e as instruções restantes do bloco básico não serão reexecutadas (com exceção das instruções de escrita na memória). Quando do reuso do bloco básico, este deverá atualizar o estado do processador, escrevendo o contexto de saída referenciado e armazenado em *Reg-Out* e *Mem-Out*, e posteriormente, passar a execução para o endereço armazenado em *Next-Block*. Um bloco básico não será reusado para os casos em que: os operandos do contexto de entrada não estão disponíveis; os valores dos operandos do contexto de entrada não coincidem com os valores dos respectivos registradores no arquivo de registradores; as operações de leitura à memória (cache) produziram valores diferentes dos armazenados em *Memo-In/Data*; as instruções de leitura à memória não obtiveram sucesso no acesso ao cache de dados *L1*; ou não existiam portas de leitura disponíveis para acesso à memória cache *L1*. É importante notar que para suportar a inclusão do mecanismo *BHB*, o processador substrato deve possuir um grande número de portas de acesso à memória cache *L1*, de modo a efetuar as leituras antecipadas das instruções de acesso à memória que serão avaliadas para validar o reuso do bloco básico. Praticamente, duas etapas de comparação serão necessárias. A primeira

validará as instruções de memória, e a segunda validará as instruções restantes do bloco básico (que incluem os valores produzidos pela primeira).

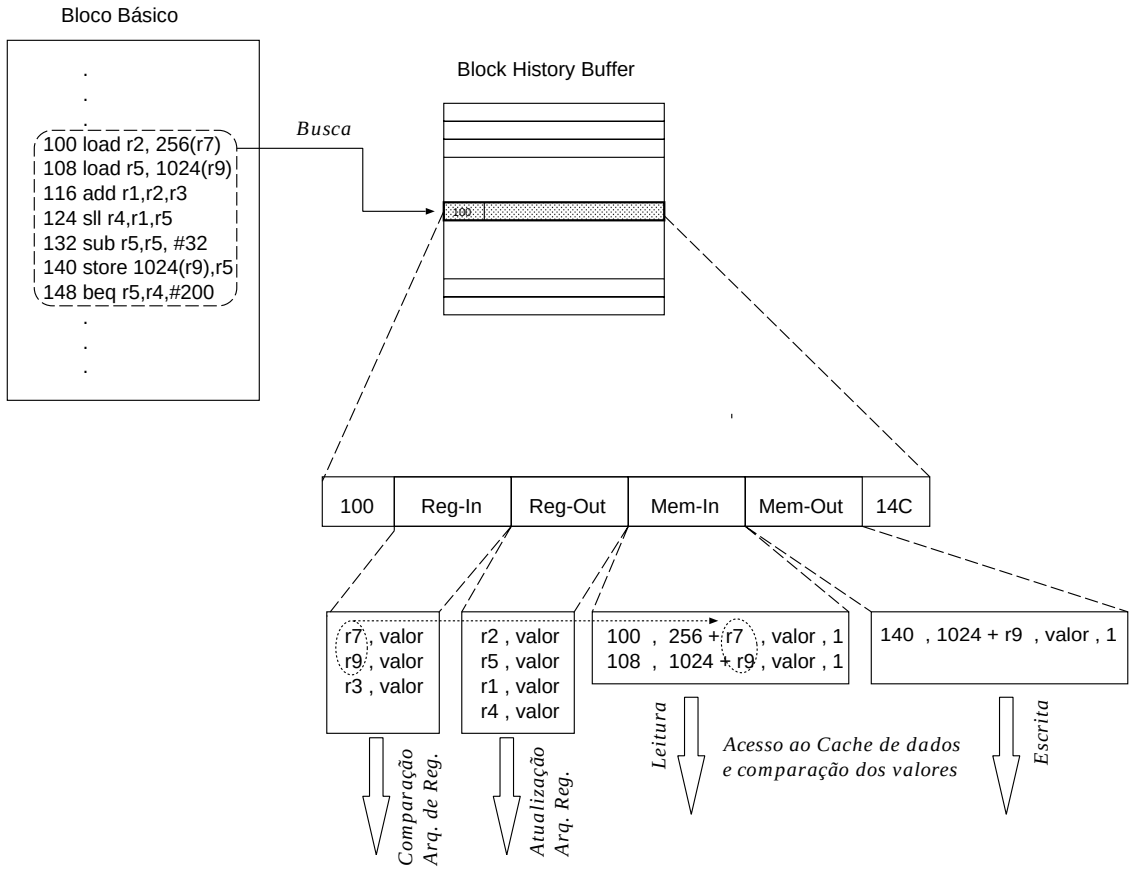


Figura 2.10: Exemplo de reuso de um bloco básico.

2.7 ALU Lookup Tables

Em [18] é avaliada a exploração de reuso aplicado diretamente a instruções que possuem altas latências de execução (em decorrência da complexidade das unidades funcionais que as executam). O esquema de reuso é muito simples e praticamente é uma reedição do mencionado em [53] (diferindo apenas no modo de acesso à tabela de memorização). O trabalho aplica e avalia memorização dinâmica para reduzir o custo de execução de instruções com múltiplos ciclos (multiplicação, divisão, raiz quadrada,etc...). Este esquema é classificado como *ID-D/ME-H/OP-H.*

Basicamente, o mecanismo funciona do seguinte modo:

(i) Os operandos e resultados de instruções particulares (instruções selecionadas por tipos e com múltiplos ciclos de latência), são armazenadas em tempo de execução, em uma tabela implementada em um cache;

(ii) A tabela é acessada em paralelo à computação convencional da instrução;

(iii) Um acesso com sucesso à tabela fornece o resultado da instrução em um único ciclo. No caso de falha no acesso não haverá penalidade no tempo de computação e a computação prosseguirá normalmente.

Como apresentado na Figura 2.11, é utilizada uma tabela de memorização denominada *Look Up Table* (*LUT*). Esta é indexada pelos valores de entrada (operandos) de cada instrução selecionada e instanciada para execução. Para cada unidade funcional que necessite de vários ciclos para a realização de sua operação específica, será associada uma *LUT* que trata as instruções selecionadas para o reuso. Para cada acesso com sucesso na *LUT*, um valor é retornado como resultado da operação (em um único ciclo), desabilitando deste modo a execução da instrução corrente na correspondente unidade funcional. No caso de uma falha na pesquisa, a unidade funcional associada continua a execução normal da instrução corrente, porém uma entrada será reservada na *LUT* para armazenar o resultado decorrente da execução da instrução (conjuntamente com os operandos da instrução). As avaliações efetuadas neste trabalho, consideraram como programas de teste, os benchmarks Multi-Media.

2.8 Trace Level Reuse

Em [30] é apresentado um estudo do potencial de reuso de instruções e traces. Neste, é definida uma estrutura de memorização denominada *Reuse Trace Memory* (*RTM*). Esta estrutura armazena informações sobre os traces selecionados idealmente para posterior reuso. A Figura 2.12(a) esboça a estrutura de uma entrada de *RTM* com tamanho ilimitado.

Descrição dos campos de uma entrada de *RTM*:

PC inicial - End. de memória da primeira instrução do trace;

Contexto de regs de entrada - Reg. e valores que instanciam o trace;

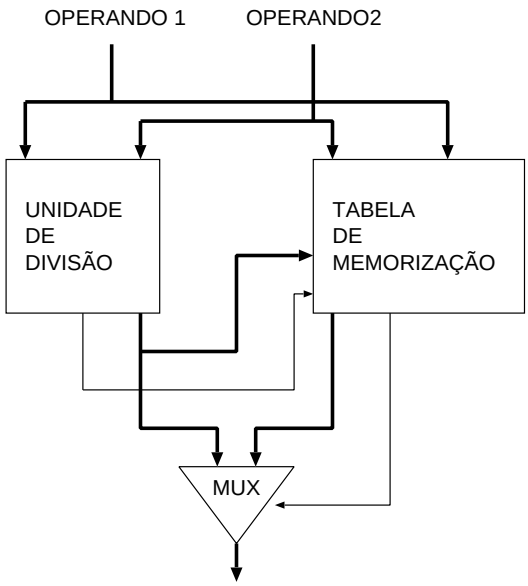


Figura 2.11: Unidade Funcional incorporando uma tabela de memorização LUT.

Contexto de mem. entrada - End. de mem. e valores que instanciam o trace;

Contexto de regs de saída - Reg. e valores produzidos pelo trace;

Contexto de mem. saída - End. de mem. e valores produzidos pelo trace;

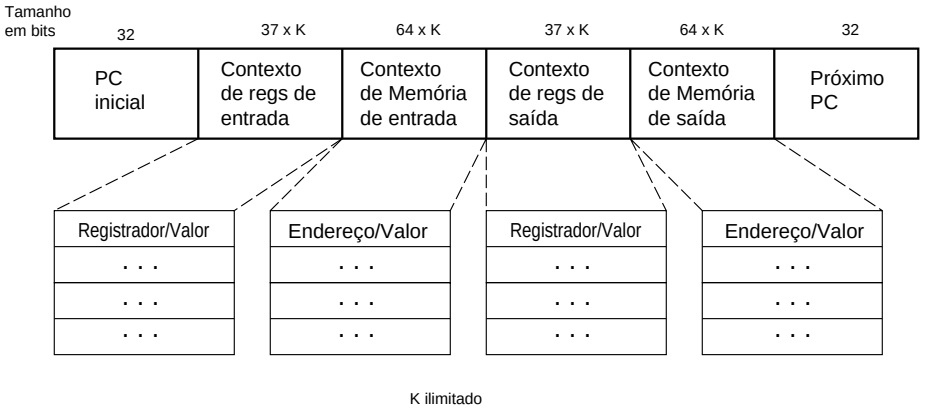
Próximo PC - End. de mem. da inst. a ser executada após o reuso do trace.

A Figura 2.12(b) apresenta o esboço de uma microarquitetura suportando a RTM e seu funcionamento.

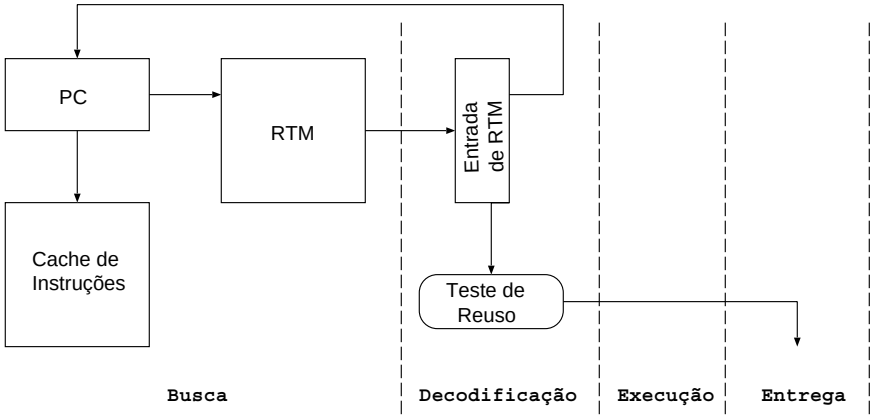
O reuso de traces é efetuado do seguinte modo: a RTM é pesquisada utilizando o endereço da instrução a ser buscada no cache de instruções. Para as entradas selecionadas em RTM ($PC\ inicial$ = endereço de busca da instrução), as informações dos contextos de entrada (registradores, posições de memória, e seus respectivos valores) são comparados com os seus respectivos pares configurados no arquivo de registradores e memória. Se a comparação efetuada verificar a igualdade dos valores comparados, então o trace é redundante e será reusado. Reusar um trace equivale a não executar as instruções que o compõe, atualizando porém, o estado do processador e da memória, através das informações do contexto de saída.

Neste trabalho são apresentados os limites superiores para redundância e aceleração que podem ser explorados pelo reuso de traces, sendo para isto assumido: uma máquina base ideal com uma janela de instruções possuindo 256 entradas; uma

RTM infinita; um mecanismo ideal para identificar e capturar os traces redundantes e um desambiguador ideal para resolução de conflitos de memória (observar que instruções possuindo efeitos colaterais são consideradas como passíveis de reuso). Apesar da avaliação do potencial de redundância e aceleração decorrentes do reuso de traces, várias considerações não foram completamente endereçadas, por exemplo: como capturar os traces redundantes, como incorporar a *RTM* em uma microarquitetura real, como manter a *RTM* consistente com os dados de memória e como manusear instruções de desvios. Este trabalho classifica-se como *ID-X/ME-H/OP-H*, onde o *X* não pode ser especificado em decorrência dos traces serem colecionados idealmente.



(a)



(b)

Figura 2.12: (a) Composição de uma entrada da RTM (b) Microarquitetura básica suportando *RTM*.

O conceito de reuso de traces foi proposto e avaliado paralelamente por este trabalho e por [20]. Enquanto neste trabalho optou-se por avaliar somente o conceito sem compromisso algum com uma eventual proposta de implementação, em [20] o conceito é avaliado levando em consideração as possíveis restrições de uma implementação real, isto é, foram feitas restrições sobre os tamanhos das tabelas de memorização, de que modo traces podem ser identificados construtivamente, quais instruções são candidatas à reuso e restrições à instruções com efeitos colaterais.

2.9 Reuso de Computações Dinâmicas Dirigidas pelo Compilador - *CRC*

Em [19] é proposta uma integração de arquitetura e compilador (hardware/software), com o objetivo de explorar a localidade de valores em grandes regiões de código. O trabalho objetiva a eliminação de computações redundantes com granularidade em nível de regiões de código.

Basicamente neste trabalho, o compilador executa uma análise para identificar prováveis regiões de código nas quais a computação pode ser reusada durante o tempo de execução. Um conjunto de instruções especializadas é incluído no conjunto de instruções básicas do processador alvo. Estas, possuem a finalidade de prover uma interface de comunicação para que o compilador forneça ao hardware as informações do contexto de entrada e saída de cada região de código reusável.

Durante o tempo de execução, os resultados da execução das regiões reusáveis são armazenados para posterior reuso. O processo de identificação de *Regiões de Computação Reusável* (*RCRs*) é baseada em compilação assistida por técnicas de *profile*, *transformação de código*, *analise interprocedural*, etc.... Além destes passos, faz-se necessário determinar a equivalência das estruturas de memória (efeitos colaterais produzidos por instruções inclusas nas regiões de código selecionadas) em intervalos regulares de tempo, afim de validar o reuso de *RCRs*.

O mecanismo proposto para explorar *RCRs* é denominado *Reuso de Computações dirigidos por Compilador* (*CCR*), e pode ser classificado como *ID-E/ME-H/OP-H*.

Este é composto pelos seguintes componentes:

(1) *CRB* - Hardware que armazena informações sobre a computação dinâmica:

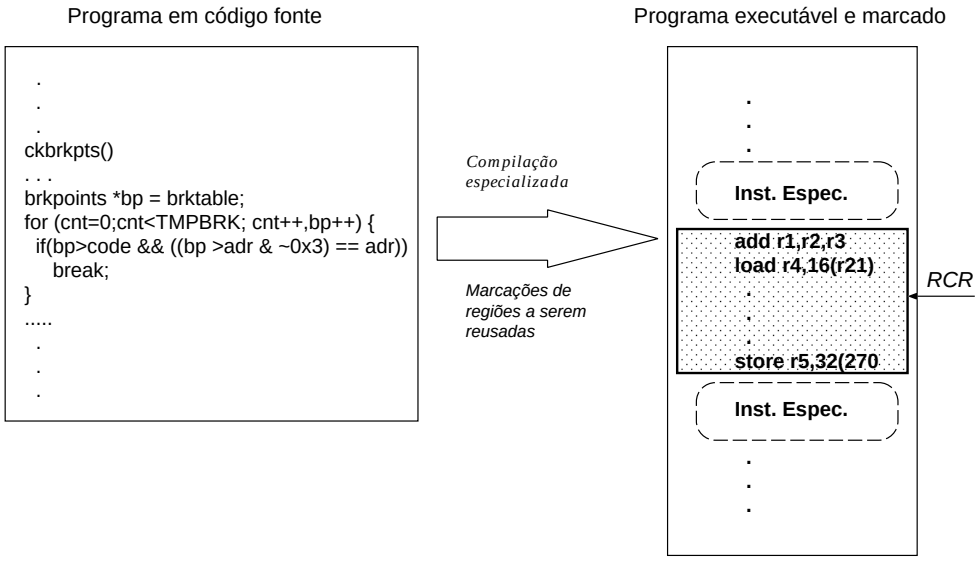


Figura 2.13: Formação e marcação de regiões de código a serem memorizadas *RCR*.

ponto de entrada; ponto de saída; e contexto de entrada e saída. Estas informações são armazenadas em uma tabela denominada *Buffer de Computação reusável - CRB*;

(2) *Instruções estendidas* - São instruções adicionadas ao conjunto de instruções básicas do processador alvo e que possuem a função de passar as informações da *RCR* para a *Microarquitetura de Reuso*;

(3) *Microarquitetura para Reuso de Computação* - Componente de hardware responsável em validar a computação armazenada em *CRB* e executar atualizações de estados arquiteturais decorrentes do reuso.

A Figura 2.13 apresenta a aplicação da compilação especializada sobre o código fonte e o código executável gerado pela transformação. Neste, são apresentadas as regiões de código que foram identificadas como provavelmente redundantes. Para as regiões identificadas, são inseridas no código objeto, instruções especializadas que irão ativar o mecanismo de reuso, bem como atualizar a estrutura de memorização (*CRB*).

O procedimento de reuso consiste nos seguintes passos:

(i) *RCRs* são identificadas durante a compilação;

(ii) Instruções especializadas são inseridas no início e fim das *RCRs*;

(iii) A execução de instruções de (ii) ativam uma pesquisa em *CRB* com o objetivo de identificar se a *RCR* corrente é reusável. Para este caso as seguintes ações são efetuadas:

(a) O contexto de entrada fornecido pelas instruções especializadas é pesquisado em *CRB*;

(b) Se a busca em (a) produzir um acerto, então a *RCR* é reusada e a *Microarquitetura para Reuso de Computação* é ativada, senão uma nova entrada em *CRB* é criada e a *RCR* corrente é inserida.

O reuso é efetuado dinamicamente por hardware e orientado por marcações efetuadas em software. As regiões delimitadas são informadas ao hardware através de instruções especializadas (mas que devem ser executadas em ordem, pois determinam uma ordem parcial à região delimitada). A Figura 2.14 apresenta a organização e estrutura da tabela de memorização ou *CRB - Computation Reuse Buffer*. Os campos de cada entrada possuem a seguinte função:

Tag - Identifica a *RCR*;

V - Indica se existe uma computação válida;

LRU - Informação para a política de atualização da *CRB*;

Instâncias de computações - Array armazenando *N* instâncias da computação;

C[i] - i-ésima instância da computação;

Entrada - Contexto de entrada *Registrador*, *Valor* e *Validade*;

Saída - Contexto de saída *Registrador*, *Valor* e *Validade*;

CI V - Indica se a computação é válida;

MEM V - Indica se as posições de memória permanecem consistentes.

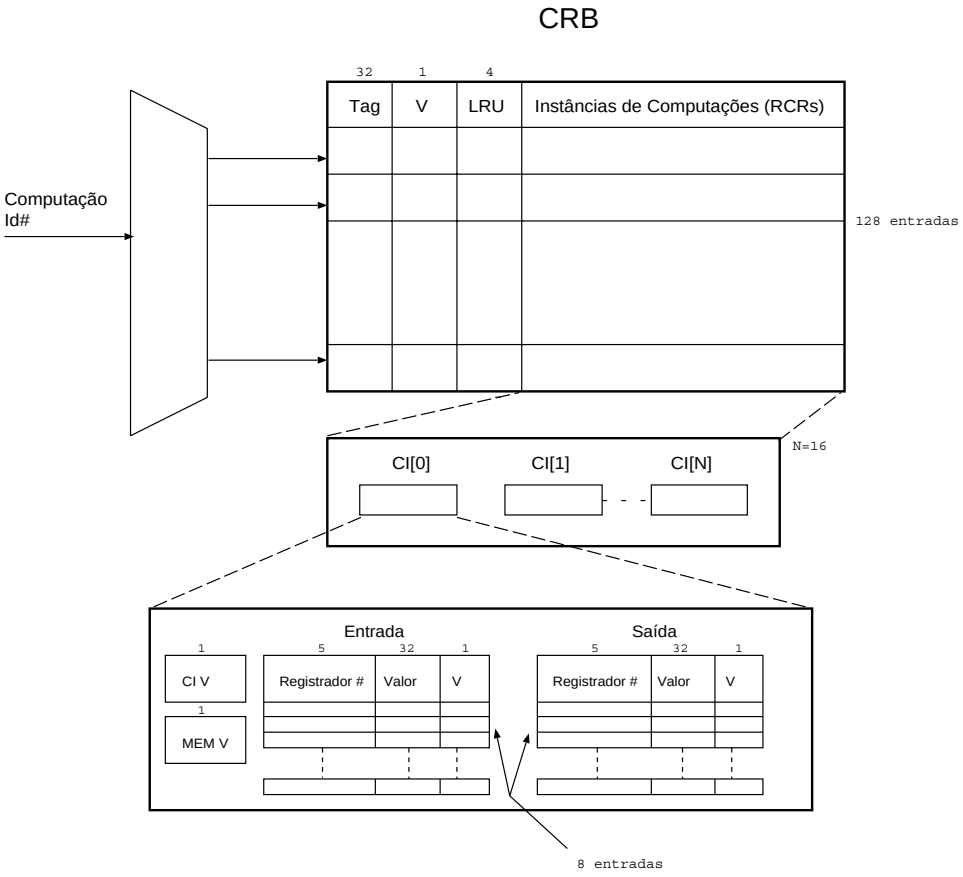


Figura 2.14: Arquitetura e composição do *CRB*.

Capítulo 3

A Memorização Dinâmica de Traces - *DTM*

Neste capítulo será apresentado um mecanismo que efetua a exploração de computações redundantes com granularidade em nível de traces. Este mecanismo será denominado *DTM*. As seções a seguir descreverão os objetivos, a estrutura básica e a descrição funcional do mecanismo *DTM*. Detalhes de cada uma de suas fases serão abordadas e exemplificadas.

3.1 Definições Preliminares

Nesta seção, serão apresentadas algumas definições necessárias para a descrição do mecanismo de ıMemorização Dinâmica de Traces - DTM.

(i) Um trace redundante é definido como uma seqüência de instruções dinâmicas e redundantes no domínio de instruções válidas;

(ii) O domínio de instruções válidas em *DTM*, representa um subconjunto do conjunto de instruções do processador que podem ser incluídas em traces;

(iii) Considerando o conjunto de instruções do *MIPS I ISA* (assumido neste trabalho como o processador alvo), as seguintes instruções não pertencem ao domínio de instruções válidas em *DTM*: instruções de *acesso a memória (LOAD/STORE)*, instruções de *ponto flutuante*, e instruções de *chamadas a rotinas do sistema operacional*;

(iv) Como *DTM* suporta execução especulativa, foi assumido que os traces deverão representar somente os caminhos de execução corretos;

(v) O contexto de entrada de um trace é definido como o conjunto de operandos fonte e os valores a eles instanciados. Os operandos fonte são referenciados por instruções que compõe o trace e são produzidos por instruções externas ao trace;

(vi) O contexto de saída de um trace é definido como o conjunto de operandos de destino e os valores a eles instanciados. Os operandos de destino são referenciados por instruções que compõe o trace, ou seja, armazenam o resultado decorrente da execução da instrução que o referencia.

3.2 Objetivos

O mecanismo de memorização dinâmica de traces *DTM*, é introduzido como uma técnica de memorização *ID-D/ME-H/OP-H*, com o objetivo de explorar computações redundantes com granularidade em nível de traces. Explorar redundância em nível de traces equivale a reusar o trace identificado como redundante e, conseqüentemente reusar de uma só vez todas as instruções componentes do trace. Um trace será reusado, se este for redundante, e será redundante quando for identificado como memorizado, ou seja, deverá estar armazenado em alguma entrada de uma tabela de memorização. Basicamente, um trace é redundante quando seu contexto de entrada (memorizado), identificado a partir da primeira instrução do trace, é idêntico ao correspondente contexto arquitetural do processador (mais especificamente, ao contexto do arquivo de registradores).

Como já mencionado anteriormente, todo mecanismo de memorização é classificado de acordo com as fases básicas de identificação de redundância (construção), memorização (ou armazenamento) e identificação de oportunidades para exploração de redundância (reuso). A Figura 3.1, esboça as fases de: (a) identificação construtiva de traces, (b) memorização de traces e (c) reuso de traces. As seções subseqüentes, descreverão estas fases considerando o mecanismo *DTM*.

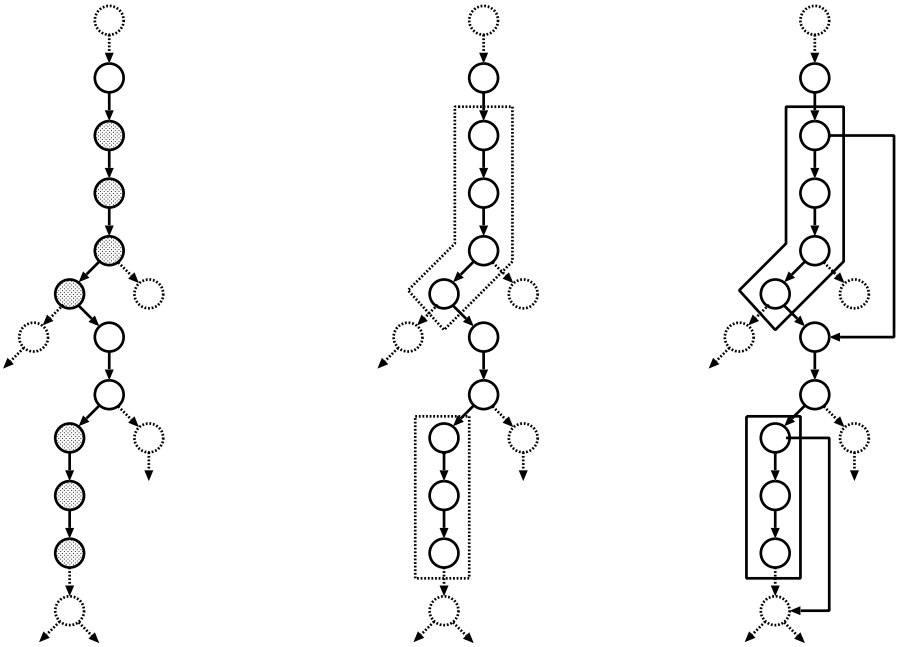


Figura 3.1: Fases básicas: (a) identificação construtiva de traces, (b) memorização de traces, (c) reuso de traces.

3.3 Identificando e memorizando traces redundantes

A identificação de traces redundantes é uma tarefa extremamente complexa. Esta complexidade decorre da própria natureza dinâmica dos fluxos de execução, sendo estes determinados pelos valores de entrada que alimentam um programa, pelos valores gerados internamente ao programa, ou ainda, por uma composição de ambos. Basicamente, um trace é redundante, se este é composto por instruções redundantes e sem efeitos colaterais. Satisfeita esta condição, pode-se construir traces, identificando-se a redundância das instruções que o compõe. Este procedimento foi adotado em [20], onde foi avaliado o potencial de redundância de traces, considerando um processador seqüencial. Este mesmo procedimento será utilizado neste trabalho (heurísticas alternativas serão alvo de trabalhos futuros).

A Figura 3.2, apresenta um exemplo de um trace que pode ser identificado como redundante, pois as instruções que o compõe são redundantes. Traces redundantes em *DTM* são delimitados por instruções não redundantes, ou não pertencentes ao

domínio de instruções válidas.

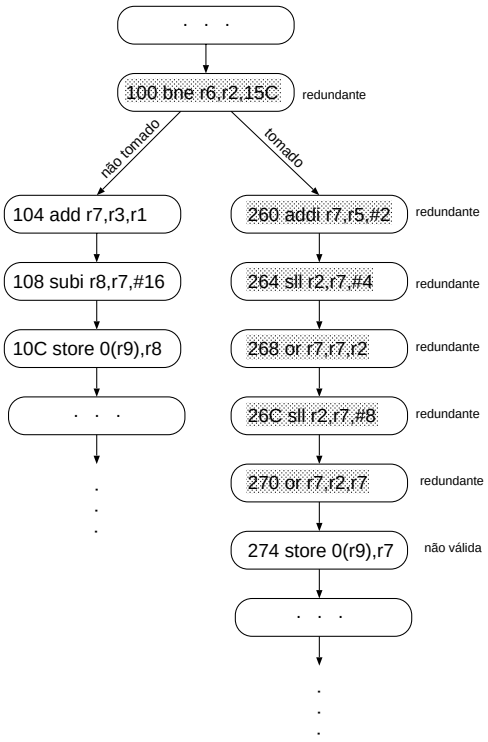


Figura 3.2: Exemplo de um trace redundante.

Além da construção de traces redundantes, estes deverão ser memorizados para posterior reuso. O procedimento de reuso deverá identificar dinamicamente, oportunidades em que um trace é redundante e se este pode ser reusado. Para este fim, deverão ser extraídas informações que promovam a identificação de traces como memorizados. Pode-se univocamente identificar traces como memorizados através do endereço de memória (pc) da primeira instrução redundante (nele encapsulada) e do contexto de entrada. Quando traces forem identificados como memorizados, estes posteriormente poderão ser reusados, isto é, as instruções que compõe o trace não necessitarão serem executadas. No entanto, ações deverão ser efetuadas de modo a atualizar o estado do processador. Esta tarefa será efetuada atualizando-se apenas os elementos do processador correspondentes aos elementos componentes do contexto de saída do trace.

Em DTM, a construção de um trace requer a identificação de redundância com granularidade em nível de instrução dinâmicas. Por sua vez, a memorização de um trace requer a obtenção de informações a partir das instruções dinâmicas redun-

dantes que são componentes do trace construído. As subseções subseqüentes irão estabelecer os requisitos básicos para a efetuação das fases de construção e memorização de traces.

3.3.1 Identificação e memorização de instruções dinâmicas

Para a construção de traces, faz-se necessário antecipadamente, identificar instruções dinâmicas[1] que são redundantes. Estas serão agrupadas em seqüências de acordo com a sua ordem de execução, de modo a formar os traces redundantes. Identificar instruções dinâmicas como redundantes equivale a verificar se já ocorreram instâncias prévias da mesma instrução. Para tal propósito, é introduzida uma estrutura de armazenamento denominada *Tabela de Memorização Global* (*Memo_Table_G*). Esta tabela possuirá a função de armazenar (memorizar dinamicamente) as instâncias dinâmicas de instruções executadas por um programa. A Figura 3.3 mostra a composição básica de uma entrada da *Memo_Table_G*. Esta é compota pelos seguintes campos com as respectivas funções:

pc - armazena o endereço de memória da instrução;

sv1 - valor do primeiro operando de entrada instanciado;

sv2 - valor do segundo operando de entrada instanciado;

res/targ - armazena o resultado de uma instrução lógica/aritmética ou o endereço de destino de um desvio incondicional/condicional, dependendo do valor dos campos *jmp* ou *brc*;

jmp - se jmp = 1, indica que a instrução é de desvio incondicional (incluindo chamadas de subrotinas e instruções de retorno);

brc - se brc = 1, indica que a instrução é de desvio condicional;

btaken - Se btaken = 1 indica que o desvio brc é tomado, se btaken = 0 indica que o desvio brc é não tomado.

[1]Durante todo o texto, os termos "instrução dinâmica", "instância dinâmica", "instrução instanciada"e "instância da instrução", serão considerados equivalentes.

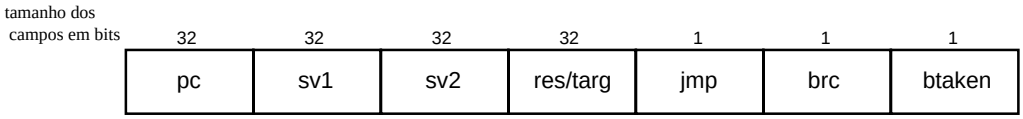


Figura 3.3: Composição de uma entrada de *Memo_Table_G*.

Durante a execução de um programa, instruções dinâmicas e pertencentes ao domínio de instruções válidas do DTM serão armazenadas (memorizadas) em *Memo_Table_G*. A Figura 3.4 esboça uma seqüência de execução e sua interação com *Memo_Table_G*. As instruções marcadas com um **X** indicam instruções não pertencentes ao domínio de instruções válidas em *DTM*.

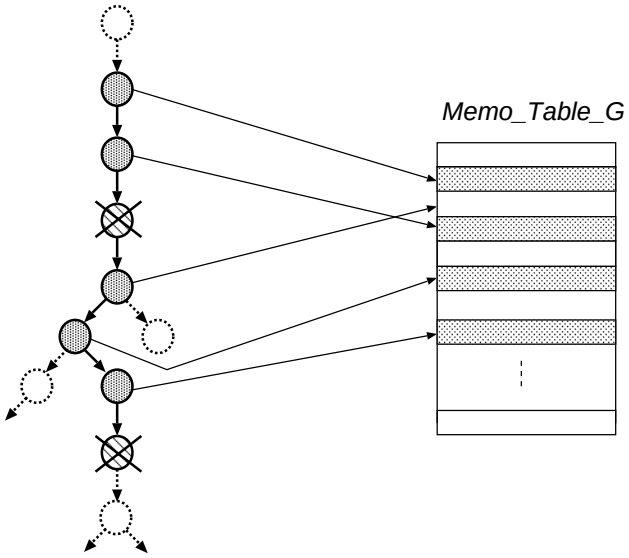


Figura 3.4: Memorização de instruções dinâmicas em *Memo_Table_G*.

3.3.2 Usando a redundância de instruções dinâmicas para construir traces

Como já mencionado anteriormente, traces serão formados por instruções dinâmicas que são redundantes. Esta subseção irá descrever incrementalmente, os passos efetuados para construção de traces. As seguintes considerações descrevem as operações

efetuadas pelo *DTM* para detectar e rotular [2] dinamicamente instruções redundantes em *Memo_Table_G*.

1. Cada entrada de *Memo_Table_G* armazenará informações sobre uma instância dinâmica de uma instrução que foi previamente executada;

2. Para cada instrução acessada e decodificada, o *DTM* verifica se esta é válida ou não. Se a instrução não pertence ao domínio de instruções válidas, então o *DTM* simplesmente rotula a instrução corrente como não redundante;

3. Para as instruções dinâmicas que pertencem ao domínio de instruções válidas, o *DTM* irá pesquisar se estas estão presentes em *Memo_Table_G*, e a busca será efetuada com base nos campos **pc**, **sv1**, **sv2**;

    3.1. Caso não exista em *Memo_Table_G* uma instância da instrução corrente, então o *DTM* marca a instrução corrente como não redundante e a insere em *Memo_Table_G*, alocando para isto, uma entrada e preenchendo seus campos com as informações da instrução correntemente instanciada;

    3.2. Se é encontrada em *Memo_Table_G* uma instância da instrução corrente, então a instrução corrente é rotulada como redundante e não é inserida em *Memo_Table_G*.

Em particular, o mecanismo *DTM* trata instruções de transferência de controle do mesmo modo que instruções aritméticas e lógicas. Em *MIPS I ISA*, o resultado de uma instrução de desvio é determinada pela comparação de dois operandos, ambos em registradores, ou um em registrador e o outro como implícito (o zero). O mecanismo *DTM* considera que uma instância de uma instrução de desvio é redundante, se existe em *Memo_Table_G* uma instância desta instrução. Instruções de desvio incondicional indireto são avaliadas a partir de seu endereço de destino, endereço este, armazenado em um registrador. Instruções de acesso à memória merecem uma atenção especial pois são compostas por duas operações (são divididas para execução): cálculo de endereço e acesso à hierarquia de memória. Para estas, somente

[2] Cada instrução receberá um rótulo (um bit) indicando se ela é redundante ou não, e este rótulo será consultado para efetivar o reuso.

será reusado o cálculo de endereços, por ser uma operação aritmética. Uma entrada de *Memo_Table_G* será alocada para uma instrução de memória, considerando apenas os operandos utilizados para o cálculo do endereço de acesso, sendo o valor resultante desta operação aritmética armazenado no campo *res/targ* e o endereço da instrução de memória será armazenada no campo *pc*.

A tabela 3.1 apresenta as alternativas de detecção de instruções redundantes em *Memo_Table_G*.

Tabela 3.1: Detecção e marcação de instruções redundantes.

Instrução válida	*Inst. presente em Memo_Table_G*	*valores dos oper. são iguais*	*Ação do DTM*
falso	não avaliado	não avaliado	instância é marcada como **não redundante**
verdadeiro	falso	não avaliado	instância corrente é marcada como **não redundante** e inserida em *Memo_Table_G*
verdadeiro	verdadeiro	falso	instância corrente é marcada como **não redundante** e inserida em *Memo_Table_G*
verdadeiro	verdadeiro	verdadeiro	instância corrente é marcada como **redundante**

Dependendo do rótulo aplicado a uma instrução instanciada, o *DTM* inicia a construção de um novo trace, inclui uma nova instrução no trace em construção (atualizando o contexto de entrada/saída), ou finaliza a construção de um trace. Se a instrução instanciada foi rotulada como não redundante, então o *DTM* encerra a construção de um trace que estiver em curso (como definido anteriormente, um trace é composto apenas por instruções redundantes que são consecutivas no fluxo de execução). Se uma instrução instanciada é marcada como redundante, então o *DTM* atualiza a informação de contexto do trace em construção, ou inicia a construção de um novo trace (no caso de não haver um trace em construção).

A Figura 3.5 esboça o procedimento de rotulação de instruções dinâmicas de acordo com sua identificação em *Memo_Table_G*, e delimita uma seqüência de instruções dinâmicas que formam um trace. Nesta figura, o fluxo de execução é representado pelo grafo e os nós correspondem a instruções dinâmicas. Cada nó hachurado corresponde a uma instrução dinâmica identificada como redundante, pois a mes-

ma encontra-se memorizada em *Memo_Table_G*. Os nós não hachurados identificam instruções não redundantes ou não pertencentes ao domínio de instruções válidas (consideradas para construção de traces). Observa-se que, após este procedimento, uma seqüência de instruções dinâmicas é identificada e será usada para construir um trace.

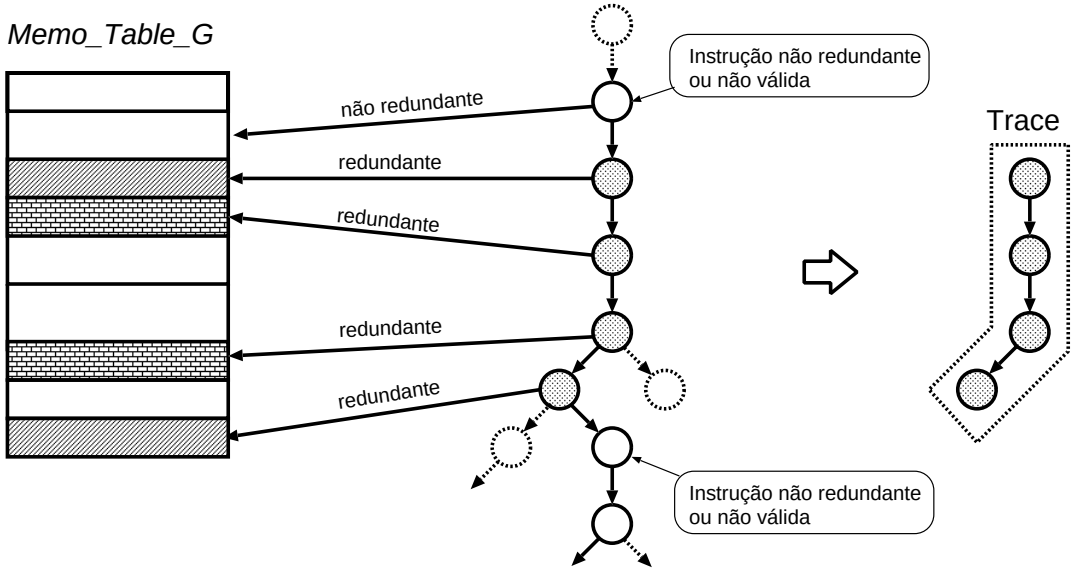


Figura 3.5: Identificando uma seqüência de instruções dinâmicas redundantes.

Como mencionado, a construção de traces envolve também a obtenção do contexto de entrada e de saída. A Figura 3.6 esboça uma estrutura empregada pelo *DTM* (buffer temporário) para a construção e armazenamento de informações sobre os contextos de traces (além de outras informações adicionais). Esta mesma estrutura representa uma entrada da tabela de memorização *Memo_Table_T*, que irá armazenar (memorizar) os traces construídos.

Os campos da estrutura esboçada na Figura 3.6, possuem a seguinte descrição:

pc - possui o endereço de memória da primeira instrução do trace;

npc - especifica o endereço da próxima instrução a ser executada para o caso em que o trace seja reusado;

icr_1,...,icr_N - indicam os registradores que armazenam o contexto de entrada;

icv_1,...,icv_N - armazenam os valores instanciados aos registradores do contexto de entrada indicados por icr_1,...,icr_N;

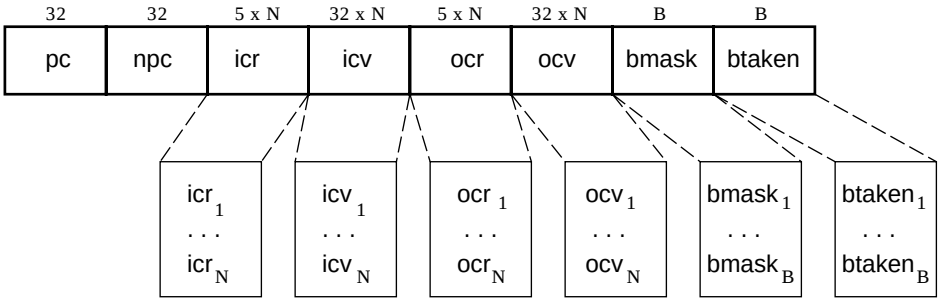


Figura 3.6: Composição das entradas da Tabela de memorização de Traces - *Memo_Table_T*.

ocr_1,...,ocr_N - indicam os registradores que armazenam o contexto de saída;

ocv_1,...,ocv_N - armazenam os valores instanciados aos registradores do contexto de saída indicados por ocr_1,...,ocr_N;

bmask - cada bit deste campo quando ativo, indica a presença de uma instrução de desvio no trace;

btaken - para cada bit ativo em *bmask*, então o bit correspondente neste campo, indica se a instrução desvio será tomada ou não tomada.

É importante notar, que o *npc* reflete todas as transferências de controle determinadas pelos desvios (jump e/ou branchs) internos ao trace. Analisando ainda a Figura 3.6, o tamanho de contexto dado por N, especifica o número máximo de elementos nos contextos de entrada e saída; o limite de desvios dado por B, indica o número máximo de instruções de desvio suportadas pelo trace.

A construção de um trace se inicia quando uma instrução dinâmica é identificada como redundante (os traces construídos possuirão no mínimo 2 instruções), e o mesmo trace é finalizado quando um dos seguintes eventos ocorrem:

1. Uma instrução não redundante é encontrada;

2. Uma instrução de acesso à memória com o cálculo de endereço redundante é encontrada, ou seja, a instrução de acesso à memória finaliza o trace por não pertencer ao domínio de instruções válidas, porém o cálculo do endereço de acesso é redundante;

3. O número de valores admissíveis no contexto de entrada ou saída tenha alcançado o limite N; ou o número de instruções de desvio incluídas no trace tenha alcançado o limite B.

Como a construção de traces é alimentada por instruções rotuladas como redundantes, a partir destas serão extraídas as informações básicas. Para a construção dos contextos de entrada e saída de traces serão considerados dois bitmaps: um mapa de contexto de entrada e um mapa de contexto de saída, onde cada bit corresponde a um registrador da arquitetura (o bit i corresponde ao registrador r_i). Cada bit do mapa de contexto de entrada/saída, indica se o registrador correspondente, possui um valor integrante do contexto de entrada/saída.

O procedimento para a formação dos contextos de entrada e de saída seguem os seguintes passos:

(i) O *DTM* ativa um bit do mapa de contexto de entrada se o respectivo registrador é um registrador fonte de uma instrução redundante incluída no trace e este é escrito por uma instrução externa ao trace, ou seja, este não se encontra marcado no mapa de contexto de saída;

(ii) Cada vez que um bit do mapa de contexto de entrada é ativado, o *DTM* adiciona o identificador e o conteúdo do registrador associado ao contexto de entrada;

(iii) O *DTM* ativa um bit do mapa de contexto de saída se o respectivo registrador é o registrador de destino de uma instrução redundante;

(iv) Cada vez que um bit do mapa de contexto de saída é ativado, o *DTM* adiciona o identificador e o conteúdo do registrador (resultado da execução da instrução) ao contexto de saída;

Para instruções de acesso à memória com cálculo de endereço redundante (estas, finalizam os traces), os operandos de entrada para cálculo do endereço de acesso seguem as condições (i),(ii),(iii) e (iv). Entretanto, o endereço calculado será armazenado como contexto de saída em algum campo ocv_I disponível e o correspondente campo ocr_I será marcado como referência a um endereço de memória calculado (não à um registrador arquitetural).

Além dos contextos de entrada e saída, outras informações adicionais são necessárias para atualizar o estado do processador para os casos em que o trace seja reusado. Estas são:

(i) Obtenção do endereço da primeira instrução dinâmica redundante. Este endereço será o endereço inicial do trace, ou seja, o endereço que o irá identificar quando da verificação de sua redundância. Este será armazenado no campo *pc* da entrada selecionada em *Memo_Table_T* para armazenar o trace;

(ii) Obtenção do endereço da próxima instrução a ser executada. Este endereço será necessário para controlar a seqüência de execução após o reuso do trace. Este endereço será o *npc* obtido a partir da última instrução redundante do trace, e será armazenado no campo *npc* da entrada selecionada em *Memo_Table_T* para armazenar o trace;

(iii) Obtenção do padrão de quebra do fluxo de execução seqüêncial. Esta informação está relacionada com as instrução de desvio que estão encapsuladas no trace. A identificação e a direção dos desvios como sendo tomado ou não tomado, serão armazenadas nos campos *bmask* e *btaken* respectivamente, e utilizados para atualizar as informações do preditor de desvios implementado por correlação (será detalhado posteriormente). As informações de direção serão obtidas a partir das instruções dinâmicas redundantes que representam os desvios e que são componentes do trace em construção.

Exemplo de construção de um trace

A Figura 3.7 ilustra um exemplo de construção de um trace (detalhado com relação a formação dos contextos). A Figura 3.7(a), retrata uma seqüência de instruções dinâmicas observada durante a execução de um programa de teste do *SPECInt95*. A partir desta seqüência, as instruções que a compõe (já instanciadas) serão pesquisadas quanto a sua ocorrência em *Memo_Table_G* (preenchida com valores em hexadecimal) como apresentado na Figura 3.7(b), e serão rotuladas como *redundantes* ou *não redundantes*.

Para cada instrução redundante (todas da seqüência no exemplo), a Figura 3.7(c) mostra como os mapas de contexto mudam dinamicamente durante a construção de um trace:

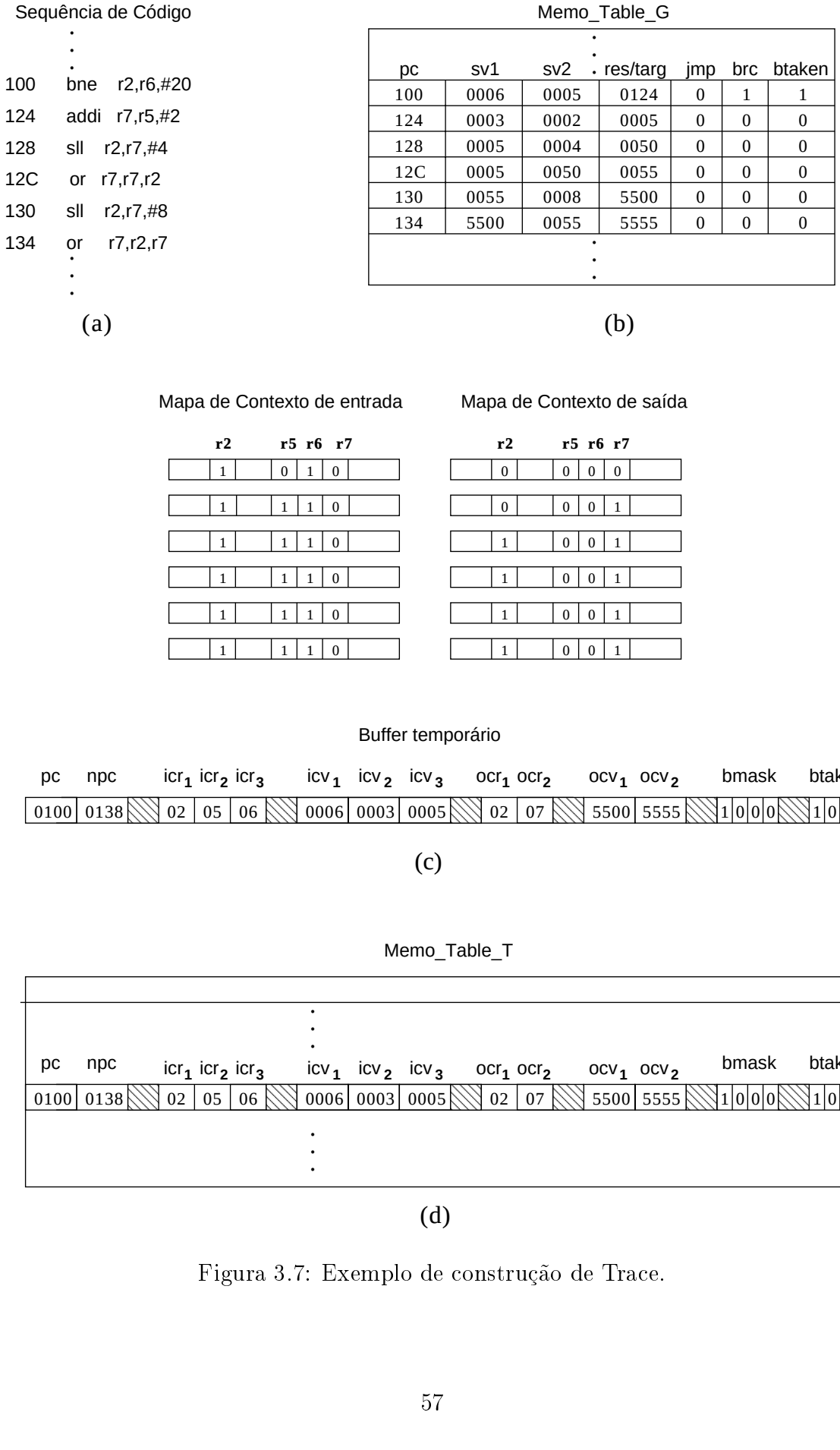


Figura 3.7: Exemplo de construção de Trace.

• A instrução *100* lê valores dos registradores *r2* e *r6*. Os bits correspondentes no mapa de contexto de entrada são setados, pois neste instante os registradores *r2* e *r6* não fazem parte do contexto de saída, isto é, seus conteúdos não foram produzidos por instruções anteriores e inclusas no mesmo trace;

• A instrução *124* lê o registrador *r5*, como o registrador *r5* não está marcado no contexto de saída, o bit correspondente no mapa de contexto de entrada será setado, indicando assim um operando externo produzido por uma instrução externa ao trace. A mesma instrução *124* escreve no registrador *r7*, logo o bit correspondente no mapa de contexto de saída será setado, e qualquer instrução (a ser incluída neste trace em formação) que efetuar uma leitura no registrador *r7*, o correspondente bit do mapa de contexto de entrada não será setado, pois o registrador *r7* já foi marcado no mapa de contexto de saída por uma instrução anterior (instrução *124* no caso apresentado), ou seja, seu conteúdo é produzido por uma instrução anteriormente incluída no trace;

• A instrução *128* utiliza o registrador *r7* como operando fonte e este já se encontra marcado como operando interno ao trace (pois está ativo no mapa de contexto de saída). Entretanto o registrador *r2* será escrito, pois é um registrador de destino da instrução, logo será marcado como ativo no mapa de contexto de saída.

Pode-se observar que após a instrução *128*, nenhum bit será setado em nenhum dos mapas de contexto. Portanto, os registradores *r2*, *r5* e *r6* irão armazenar os valores do contexto de entrada do trace (e serão determinantes na identificação de um trace redundante), e os registradores *r2* e *r7* irão armazenar os valores do contexto de saída (determinantes no reuso do trace, pois irão atualizar o contexto do processador). Pode-se observar a partir do exposto, que o *DTM* determina se um operando (registrador) é produzido por alguma instrução externa ao trace, verificando o estado do bit corresponde ao operando no mapa de contexto de entrada.

As informações de desvio de fluxo de execução correspondente a cada instrução redundante identificada em sua respectiva entrada de *Memo_Table_G*, o endereço inicial (fornecido pelo *pc* da primeira instrução redundante) e o endereço final do trace construído (fornecido pelo *npc* da última instrução redundante), serão transferidas diretamente para o buffer temporário (estrutura que armazena o trace em construção). A Figura 3.7(c) esboça o armazenamento das informações do trace

construído, a partir das informações extraídas pelo procedimento de construção. Após este procedimento, o trace construído e armazenado no *buffer temporário* será transferido para uma entrada em *Memo_Table_T* como exposto pela Figura 3.7(d).

A Figura 3.8 reforça visualmente o procedimento de construção de traces. Nesta, é apresenta a seqüência de coleta de informações a partir de uma seqüência de instruções redundantes que irão formar um trace e o preenchimento do *buffer temporário* para posterior armazenamento em alguma entrada de *Memo_Table_T*.

3.4 Reusando traces e instruções redundantes

Reusar traces, produzirá um efeito equivalente a se reusar uma seqüência de instruções dinâmicas, de modo que todas as instruções representadas no trace serão reusadas de uma só vez. Para realizar este objetivo, torna-se necessário identificar oportunidades que permitam tal operação. No decorrer desta seção serão explanadas as ações que permitem determinar dinamicamente a reusabilidade de traces.

Concorrentemente à construção de traces, o *DTM* procura por instruções redundantes e traces redundantes que possam ser reusados. As operações efetuadas pelo *DTM* para detectar instruções redundantes foram descritas na seção anterior. Esta seção, mostra como o *DTM* detecta e reusa traces e instruções redundantes.

A seguir, será descrito o procedimento para identificação e reuso de traces e instruções redundantes:

1. O *DTM* verifica para cada instrução a ser acessada no cache de instruções, se esta é uma instrução inicial de algum trace previamente armazenado em *Memo_Table_T*. Inicialmente isto é efetuado comparando-se o endereço da instrução acessada com o campo *pc* de cada entrada de *Memo_Table_T*;

    1.1. Se a comparação efetuada não identificar nenhuma entrada em *Memo_Table_T*, então a busca não identificou nenhum trace redundante;

    1.2. Se a comparação efetuada identificar entradas em *Memo_Table_T*, então o *DTM* seleciona todas as entradas de *Memo_Table_T* que satisfaçam a comparação. Para estas entradas, serão comparados os valores armazenados nos registradores do arquivo de registradores indicados por icr_1,...,icr_N com o respectivo contexto armazenado em icv_1,...,icv_N:

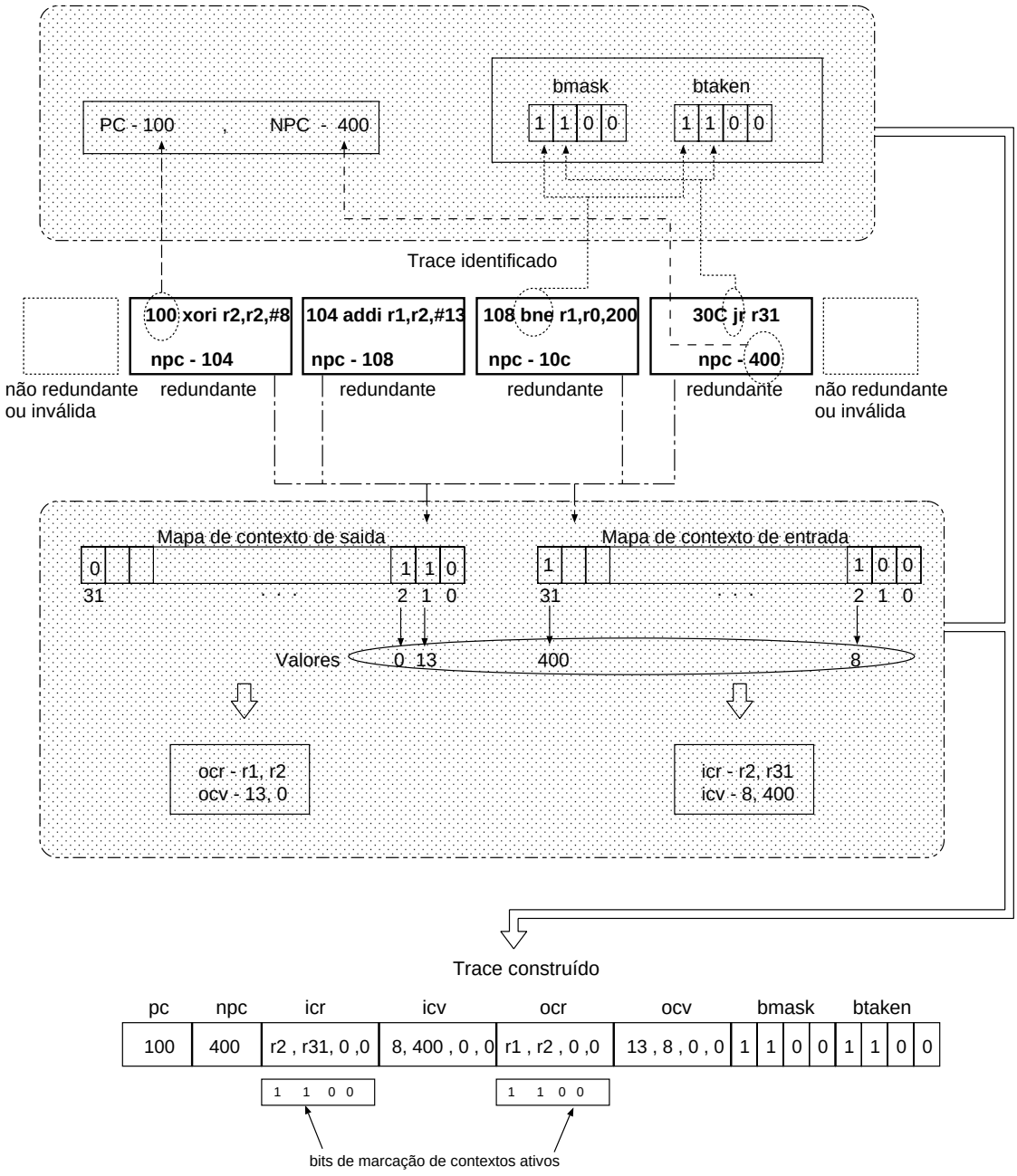


Figura 3.8: Extração de informações obtidas a partir da seqüência de instruções rotuladas como redundantes.

1.2.1. Se a comparação efetuada identificar o contexto arquitetural (valores armazenados nos registradores do processador indicados pelos campos icr_1,...,icr_N de uma entrada em *Memo_Table_T*) idêntico ao contexto de entrada de algum trace selecionado, então será identificado um trace redundante;

1.2.2 Senão, nenhum trace redundante foi identificado.

É imediato observar que as operações para detectar instruções redundantes e traces redundantes são análogas, a única diferença refere-se a estrutura acessada. Instruções redundantes são detectadas através de *Memo_Table_G*, enquanto traces redundantes são detectados via *Memo_Table_T*. Um acesso com sucesso ocorre em *Memo_Table_G*, se uma instrução redundante é detectada. Similarmente, um acesso com sucesso ocorre em *Memo_Table_T*, se um trace redundante é encontrado. Como mostra a tabela 3.2, o *DTM* decide que informações reusar, de acordo com a ocorrência de acessos com sucesso nas tabelas de memorização.

Tabela 3.2: Reusando Instruções ou Traces.

Evento em Memo_Table_G	*Evento em Memo_Table_T*	*Ação*
falha	falha	Não há instruções ou trace redundantes continue a execução normalmente.
falha	acerto	Um trace redundante é reusado.
acerto	falha	Uma instrução redundante é reusada.
acerto	acerto	Apenas o trace redundante é reusado.

Se ocorre um acesso com sucesso apenas em *Memo_Table_G*, então uma instrução redundante é reusada. No caso de instruções aritméticas/lógicas redundantes, o resultado reusado será obtido do campo *res/targ* da correspondente entrada selecionada em *Memo_Table_G*. No caso de instruções de desvio redundantes, os campos *res/targ* e *btaken* serão utilizados respectivamente, para redirecionar a unidade de busca do processador e atualizar o estado do preditor de desvios. Se ocorrer um acesso com sucesso em ambas tabelas de memorização, o *DTM* prioriza o reuso das várias instruções cobertas pelo trace ao invés de reusar uma instrução simples. Neste caso, os registradores do processador referenciados pelos campos ocr_1,...,ocr_N,

serão atualizados pelos valores armazenados nos campos ocv_1,...,ocv_N do contexto de saída a ser reusado. O *DTM* carrega o contador de programa do processador com o endereço armazenado no campo *npc*, pulando portanto, a execução das instruções cobertas pelo trace. Em adição, o *DTM* atualiza o estado do preditor de desvios usando a informação armazenada nos campos *bmask* e *btaken*.

A Figura 3.9 apresenta um exemplo de reuso de um trace a partir de uma seqüência de código. Neste exemplo, a *Memo_Table_T* é interrogada pelo endereço *100* e retorna as entradas para as quais o referido endereço ocorre no campo *pc* (duas entradas no exemplo). Para as entradas selecionadas, o contexto de entrada é comparado com os respectivos registradores no arquivo de registradores. No exemplo, a primeira entrada selecionada apresenta o contexto de entrada satisfeito pela comparação (contrariamente, a segunda não satisfaz), logo o trace é redundante e será reusado. Os passos subseqüentes para reusar o trace irão atualizar o contexto de saída a partir das informações da entrada selecionada (como determinado pelo procedimento descrito anteriormente). O reuso de instruções simples será análogo ao procedimento para reuso de traces.

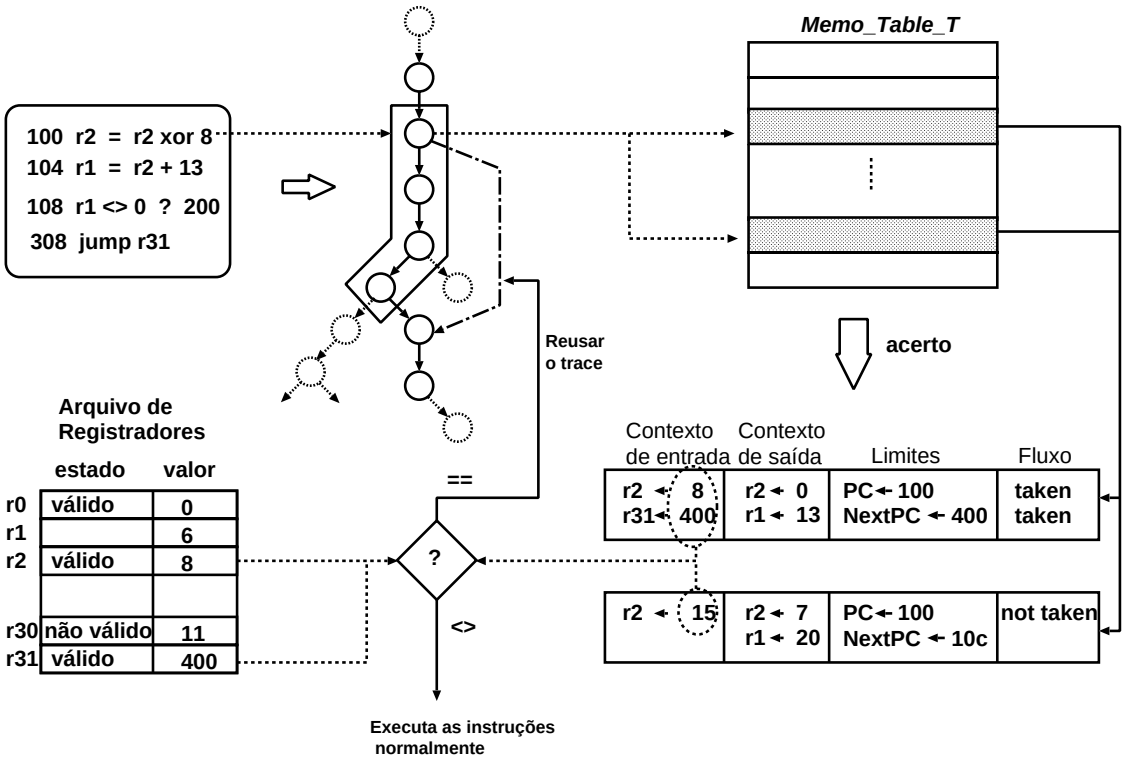


Figura 3.9: Exemplo de reuso de trace.

Capítulo 4

Uma Arquitetura Superescalar incorporando o mecanismo *DTM*

O capítulo anterior foi introduzido para retratar a funcionalidade do mecanismo *DTM*. O presente capítulo descreve como o mecanismo *DTM* pode ser adicionado a uma microarquitetura superescalar. No decorrer das seções, serão descritos e detalhados os problemas encontrados, as soluções implementadas e a interação entre o mecanismo e a microarquitetura do processador superescalar substrato.

4.1 A Microarquitetura Superescalar Substrato

Esta seção descreve a microarquitetura superescalar que será utilizada para suportar a inclusão do *DTM*.

A Figura 4.1, retrata a microarquitetura superescalar que será considerada como substrato para o mecanismo *DTM*. Esta microarquitetura emprega uma variante do algoritmo de Tomasulo [67] para suportar execução fora de ordem, e um buffer de reordenação [60] para suportar execução especulativa. Exceto por poucas diferenças (esquema de tagging), a microarquitetura é similar à incorporada no microprocessador *K6 3D* [66]. Serão descritos apenas os aspectos operacionais da microarquitetura substrato que são associados ao mecanismo *DTM*.

O *Estágio de Busca* considerando um endereço recebido, lê instruções do cache de instruções. Considerando o mesmo endereço enviado ao cache de instruções, o *Preditor de Desvios* efetua as predições sobre as instruções de desvio. Finalizando as

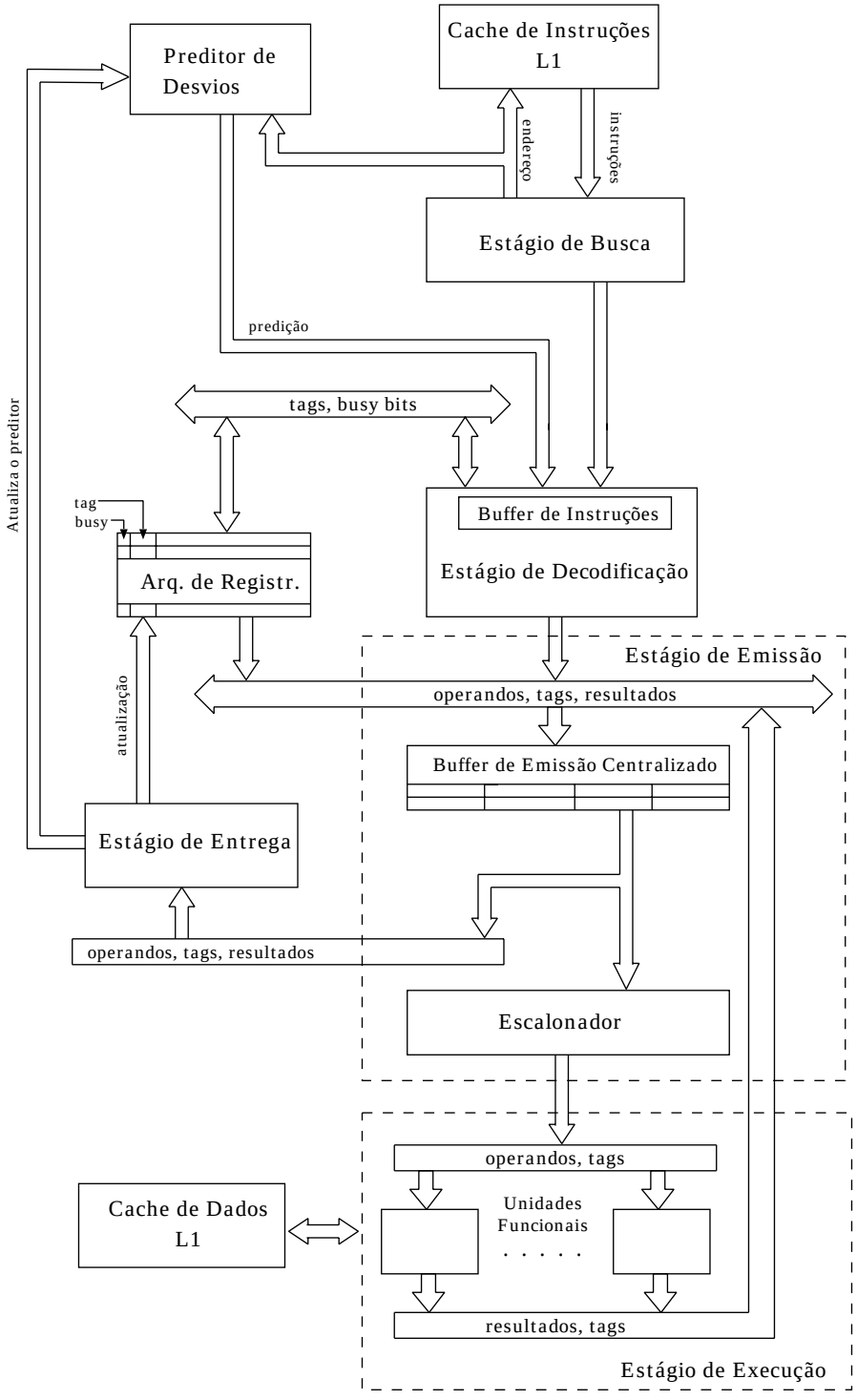


Figura 4.1: Microarquitetura Superescalar Substrato.

atividades neste estágio, as instruções acessadas e preditas serão transferidas para o *Buffer de Instruções*. É importante ressaltar que serão desconsideradas as instruções subseqüentes a uma instrução de desvio predita como tomada.

Para cada instrução no *Buffer de Instruções*, o *Estágio de Decodificação* decodifica a instrução e lê os registradores fonte e seus busy bits. Se um busy bit está inativo, então o registrador fonte correspondente possui um dado válido; do contrário, o registrador fonte possui um dado inválido. No último caso, o tag associado ao registrador em questão especifica a instrução que produzirá o operando.

O *Estágio de Decodificação* despacha (em ordem) as instruções decodificadas para as estações de reserva. Os dados válidos também serão enviados para a estação de reserva, porém, caso os dados sejam inválidos, tags serão enviados em seu lugar. Durante o despacho de uma instrução, o *Estágio de Decodificação* ativa o busy bit do registrador destino e seta a tag para o número da estação de reserva destinatária. A largura de despacho w, define o número máximo de instruções que podem ser despachadas em um mesmo ciclo pelo *Estágio de Decodificação*.

O *Buffer de Emissão* centraliza todas as estações de reserva[1] e fornece a funcionalidade do *Buffer de Reordenação* [62]. O *Buffer de Emissão* opera como uma *FIFO*: instruções despachadas são colocadas em uma próxima entrada disponível no fim do buffer, enquanto instruções executadas são retiradas das entradas ocupadas e iniciais do buffer.

O *Estágio de Emissão* executa o escalonamento dinâmico de instruções, procurando no *Buffer de Emissão* por instruções aptas à execução. O escalonamento leva em consideração a disponibilidade de operandos e unidades funcionais livres, porém é independente da posição da instrução no *Buffer de Emissão*. Uma vez que uma instrução tenha sido processada pela unidade funcional, o *Estágio de Execução* reenvia (em um próximo ciclo) o resultado da instrução para o *Estágio de Emissão*. O *Estágio de Emissão* armazena o resultado na estação de reserva, mantendo a instrução finalizada. Baseado na ordenação das instruções no *Buffer de Emissão* e nos tags utilizados, o *Estágio de Emissão* detecta dependências entre instruções e reenvia dados das instruções precedentes (iniciais no buffer) para instruções dependentes

[1] Uma entrada no *Buffer de Emissão* e uma estação de reserva representam o mesmo elemento da arquitetura.

(posteriores as iniciais no buffer).

O *Estágio de Execução* efetua a execução das instruções escalonadas em suas unidades funcionais. As unidades funcionais incluem unidades para processamento de valores inteiros, ponto flutuante, operações de memória (disponibiliza uma porta de acesso à hierarquia de memória) e resolução de desvios.

O *Estágio de Entrega* atualiza o arquivo de registradores na ordem seqüencial do programa. Uma instrução executada será entregue se todas as condições abaixo forem satisfeitas:

(1) Está em uma entrada frontal do *Buffer de Emissão*;

(2) Se todas as instruções em entradas frontais e precedentes no *Buffer de Emissão* já foram entregues;

(3) Não é subseqüente a um desvio com predição não confirmada.

O *Estágio de Entrega* escreve o resultado da instrução a ser entregue em um registrador do arquivo de registradores que possua tag correspondente ao tag da instrução entregue e reseta o correspondente busy bit. Dado que w instruções frontais do *Buffer de Emissão* tenham sido entregues, o *Estágio de Entrega* irá removê-las do *Buffer de Emissão*.

O *preditor de desvios* efetua predições considerando os mesmos endereços enviados pelo *Estágio de Busca* para o cache de instruções. As predições efetuadas são transferidas com as respectivas instruções acessadas para o *Buffer de Instruções*. Predições são finalizadas ao se detectar o primeiro desvio predito como tomado. Instruções de desvio que são preditas, possuirão sua predição validada ou não, após a sua execução, e enquanto não avaliadas, as instruções subseqüentes serão executadas especulativamente. Para o caso em que a predição seja confirmada como correta, todas as instruções subseqüentes (excetuando-se as também preditas), serão marcadas como não especulativas e poderão ser entregues normalmente pelo *Estágio de Entrega*. Quando a predição efetuada for avaliada como incorreta, as instruções subseqüentes à instrução de desvio predita, serão descartadas e o fluxo de busca será redirecionado para o destino correto. O *preditor de desvios* é atualizado no *Estágio de Entrega*, evitando assim atualizações especulativas. Predições de desvios incondicionais indiretos (dependem do valor de um registrador), farão acessos a uma *BTB*

(*Branch Target Buffer*), com o objetivo de identificar especulativamente o endereço alvo do desvio.

4.2 Adicionando o mecanismo *DTM*

Nesta seção, o mecanismo *DTM* será adicionado à microarquitetura substrato apresentada na seção anterior. O mecanismo *DTM* aqui descrito, é organizado em três estágios, e serão referenciados como **DS1**,**DS2** e **DS3**. Estes estágios acessam ambas tabelas de memorização para construir, memorizar, e reusar instruções simples e traces. Na Figura 4.2 é apresentada o esboço da microarquitetura substrato suportando o mecanismo *DTM*, os três estágios do *DTM* e seus *datapaths* (aparecem em linhas tracejadas).

A tabela 4.1, lista as principais operações do mecanismo *DTM* e a função de cada um dos estágios.

Tabela 4.1: Correspondência entre as operações do *DTM* e seus estágios.

Op	*Função*	*Estágio DTM*
	Construção de Trace	
(1)	Insere instâncias de instruções em *Memo_Table_G*.	**DS1**,**DS3**
(2)	Inicia a seleção de instruções redundantes em *Memo_Table_G*.	**DS1**
(3)	Identifica instâncias de instruções redundantes.	**DS2**
(4)	Atualiza o mapa de bits e preenche os buffers temporários.	**DS3**
	Reuso de Traces ou Instruções	
(2)	Inicia a seleção de instruções redundantes em *Memo_Table_G*.	**DS1**
(3)	Identifica instâncias de instruções redundantes.	**DS2**
(5)	Inicia a seleção de traces redundantes em *Memo_Table_T*.	**DS1**
(6)	Identifica traces redundantes.	**DS2**
(7)	Decide se uma instrução ou um trace podem ser reusados.	**DS2**

As operações relacionadas ao reuso de instruções e construção de traces são efetuadas pelo mecanismo *DTM* da seguinte forma:

• **Op(1)**- Durante o ciclo de busca, o estágio **DS1** aloca uma entrada em *Memo_Table_G* para cada instrução acessada e preenche o campo *pc* de cada entrada com o endereço da instrução alocada para a respectiva entrada. Durante o ciclo de decodificação, o estágio **DS2** fornece ao *Estágio de Decodificação*, o índice de cada entrada de *Memo_Table_G* alocada no ciclo anterior. O *Estágio de Decodificação*

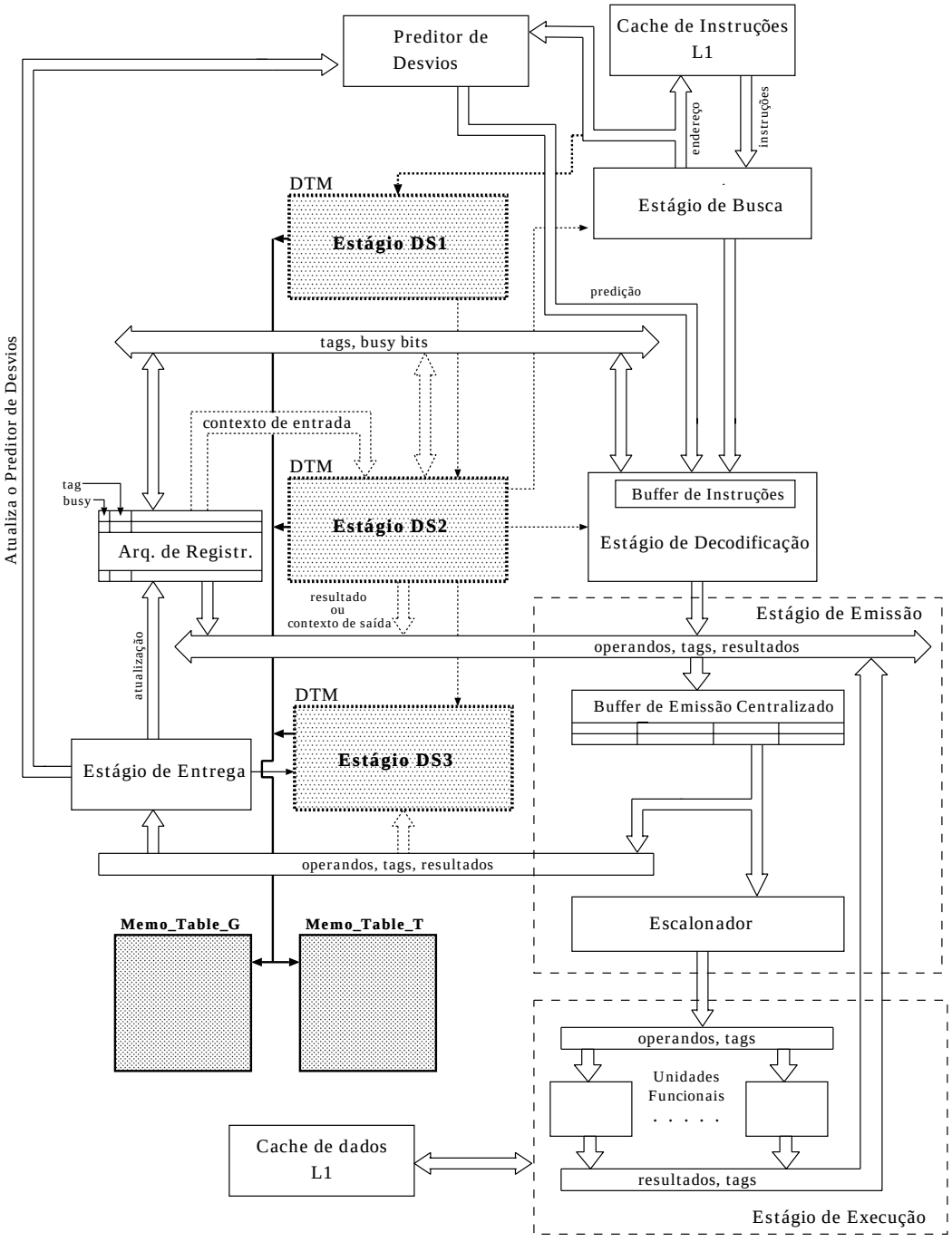


Figura 4.2: Microarquitetura suportando o mecanismo *DTM*.

anexa os índices às correspondentes instruções despachadas. Durante a entrega de uma instrução, o *Estágio de Entrega* fornece os operandos da instrução, o resultado da instrução e o índice de *Memo_Table_G* para o estágio **DS3**. O estágio **DS3** por sua vez, preenche os campos *sv1*,*sv2* e *res/targ* da entrada de *Memo_Table_G* indicada pelo índice recebido. Se a instrução entregue for um jump ou um branch, então os campos *jmp*, *brc* e *btaken* serão preenchidos apropriadamente. Sempre que o *Estágio de Entrega* descartar instruções seguintes a um desvio predito incorretamente, o estágio **DS3** invalidará todas as entradas alocadas (entretanto não inicializadas) em *Memo_Table_G*. Deste modo o mecanismo *DTM* captura apenas instruções presentes no caminho de execução correto, como estipulado na definição do mecanismo *DTM*;

• **Op(2)**- Durante o ciclo de busca, o estágio **DS1** recebe o endereço de busca fornecido pelo *Estágio de Busca*. O estágio **DS1** então compara o endereço de cada instrução (com destino ao *Buffer de Instruções*) com o conteúdo do campo *pc* de cada entrada de *Memo_Table_G* e seleciona apenas as entradas cuja comparação resultou em sucesso. Apenas as entradas pré-selecionadas serão referenciadas durante a busca por instruções redundantes;

• **Op(3)**- Durante o ciclo de decodificação, o *Estágio de Decodificação* efetua a leitura dos operandos fonte (registradores e seus respectivos busy bits) das instruções armazenadas no *Buffer de instruções* (para posterior despacho). Esta leitura fornece também para o estágio **DS2**, os mesmos valores necessários para a verificação de redundância sobre as entradas pré-selecionadas. O estágio **DS2** compara os valores recebidos com os respectivos campos *sv1* e *sv2* de cada entrada pré-selecionada de *Memo_Table_G*. Vale ressaltar que a comparação descrita só será efetivada caso os busy bits não estejam ativos, ou seja, caso os valores armazenados nos registradores sejam válidos. O resultado das comparações indicarão se as instruções sendo decodificadas são redundantes ou não. Para o caso em que a instrução seja redundante, o resultado da instrução redundante é despachado para o *Buffer de Emissão* e a estação de reserva destinatária é marcada como possuindo uma instrução redundante (rotulada). Será explicado mais a frente que instruções redundantes não serão emitidas para execução, entretanto elas serão entregues normalmente;

• **Op(4)**- Durante a entrega de uma instrução, o *Estágio de Entrega* fornecerá

ao estágio **DS3**, os nomes dos registradores fonte e seus respectivos valores instanciados, o nome do registrador de destino e o resultado a ele instanciado decorrente da execução da instrução. Dependendo da rotulação da instrução como sendo redundante ou não (indicado previamente pelo estágio **DS2**), o estágio **DS3** decidirá em iniciar a construção de um novo trace, ou atualizar o contexto de entrada e saída de algum trace em construção, ou ainda, finalizar um trace em fase de construção (ver tabela 3.1). Utilizando-se da informação fornecida pelo *Estágio de Entrega*, o estágio **DS3** atualizará os mapas de contexto e preencherá os campos *icr* ,*icv* ,*ocr* e *ocv* do buffer temporário. Se a instrução entregue é um desvio, então o bit corrente do campo *bmask* será ativado e o bit *btaken* correspondente será setado se o desvio for tomado. Quando um trace em construção for finalizado, os campos *pc* e *npc* serão preenchidos respectivamente com o endereço da primeira instrução do trace e com o endereço da instrução imediatamente subseqüente a última instrução do trace (o *npc* da última instrução do trace).

Considerando o reuso de traces, o mecanismo *DTM* executa as operações descritas a seguir:

• **Ops(2)**,**(3)**- Estas duas operações são comuns para construção de traces e para reuso, e já foram descritas anteriormente;

• **Op(5)**- Durante o ciclo de busca, o estágio **DS1** recebe o endereço de busca fornecido pelo *Estágio de Busca*. O estágio **DS1** então compara o endereço de cada instrução (com destino ao *Buffer de Instruções*) com o conteúdo do campo *pc* de cada entrada de *Memo_Table_T*. Apenas as entradas de *Memo_Table_T* que coincidirem com o conteúdo do campo *pc*, serão selecionadas para participarem na busca por traces redundantes. Para cada entrada acessada, os respectivos campos icr_1,...,icr_N são lidos e identificados;

• **Op(6)**- Durante o ciclo de decodificação, o *Estágio de Decodificação* irá efetuar uma requisição de leitura ao arquivo de registradores, para que este leia os operandos fonte (registradores e seus respectivos busy bits) das instruções armazenadas no *Buffer de instruções* (para posterior despacho). Anexada a esta requisição de leitura, serão incorporados os registradores especificados pelos campos icr_1,...,icr_N de todas as entradas pré-selecionadas de *Memo_Table_T* (em **Op(5)**) . Após a leitura, o estágio **DS2** então compara os valores armazenados nos registradores arquiteturais

especificados pelos campos icr_1,...,icr_N aos respectivos valores armazenados nos campos icv_1,...,icv_N das mesmas entradas pré-selecionadas em *Memo_Table_T*. O resultado da comparação indica se existe ou não, um trace redundante. Para o caso em que exista um trace redundante, após este procedimento, tal trace já se encontra identificado e poderá ser reusado. Será explicado mais adiante quais as ações efetuadas para efetivar o reuso de traces;

• **Op(7)**- O resultado das operações **Op(3)** e **Op(6)** podem indicar a ocorrência de um acerto em *Memo_Table_G* e *Memo_Table_T* respectivamente. Se não ocorrer um acerto em *Memo_Table_G* e *Memo_Table_T*, então o estágio **DS2** indica ao *Estágio de Decodificação*, que todas instruções podem ser despachadas normalmente. Se ocorrer um acerto em ambas tabelas de memorização, então o estágio **DS2** opta por reusar uma instrução ou um trace (de acordo com a tabela 3.2).

Se uma instrução aritmética/lógica é redundante, o estágio **DS2** lê o resultado da instrução em *Memo_Table_G* e fornece-o para o *Estágio de Decodificação* [2], este envia o resultado para uma estação de reserva. Uma flag na estação de reserva, indica que a mesma possui o resultado de uma instrução reusada (rotulada como redundante). Estando presente no *Buffer de Emissão*, o resultado pode ser normalmente repassado para as instruções dependentes (em um próximo ciclo). Estações de reserva armazenando instruções reusadas não as emite para execução, porém elas são retiradas normalmente pelo *Estágio de Entrega*. Entregar o resultado de uma instrução reusada, significa escrever o resultado no registrador de destino da instrução. Se um desvio condicional é reusado e a predição efetuada anteriormente difere da predição armazenada na correspondente entrada (campo *taken*) de *Memo_Table_G*, então o estágio **DS2** redireciona o *Estágio de Busca* para o endereço de destino. A instrução é despachada para o *Buffer de Emissão*, apenas com o propósito de permitir que o *Estágio de Entrega* complete a execução em ordem e que interrupções precisas sejam suportadas [60].

Se um trace é reusado, então o estágio **DS2** lê na *Memo_Table_T*, as informações do trace redundante e fornece o contexto de saída do trace constituído pelos pares <ocr_1,ocv_1>,<ocr_N,ocv_N>, para o *Estágio de Decodificação*. Em adição, o estágio **DS2** redireciona o *Estágio de Busca* para o endereço especificado pelo campo *npc*

[2] Este estágio também é responsável pelo despacho de instruções para o *Buffer de Emissão*.

(será discutido na próxima seção). O *Estágio de Decodificação* despacha cada par $<ocr_i, ocv_i>$ para uma estação de reserva. Estando no *Estágio de Emissão*, os valores $ocv_1,\ldots,ocv_N$ podem ser repassados para as instruções dependentes (em um próximo ciclo). Estações de reserva armazenando os dados do contexto de saída são marcadas como possuindo instruções reusadas e são manuseadas como mencionado para instruções simples, embora elas não sejam escalonadas para execução, elas são retiradas normalmente do *Buffer de Emissão*. Entregar o contexto de saída equivale a escrever cada valor $ocv_1,\ldots,ocv_N$ nos registradores de destino especificados por $ocr_1,\ldots,ocr_N$.

A seqüência de figuras a seguir, esboçam a organização das etapas descritas anteriormente.

A Figura 4.3 apresenta os passos efetuados pelo estágio **DS1**, onde são iniciadas as buscas por instruções ou traces redundantes. Nesta etapa, as tabelas de memorização são interrogadas objetivando uma pré-seleção das entradas candidatas. Os passos são descritos pelas operações **Op(2)** e **Op(5)**.

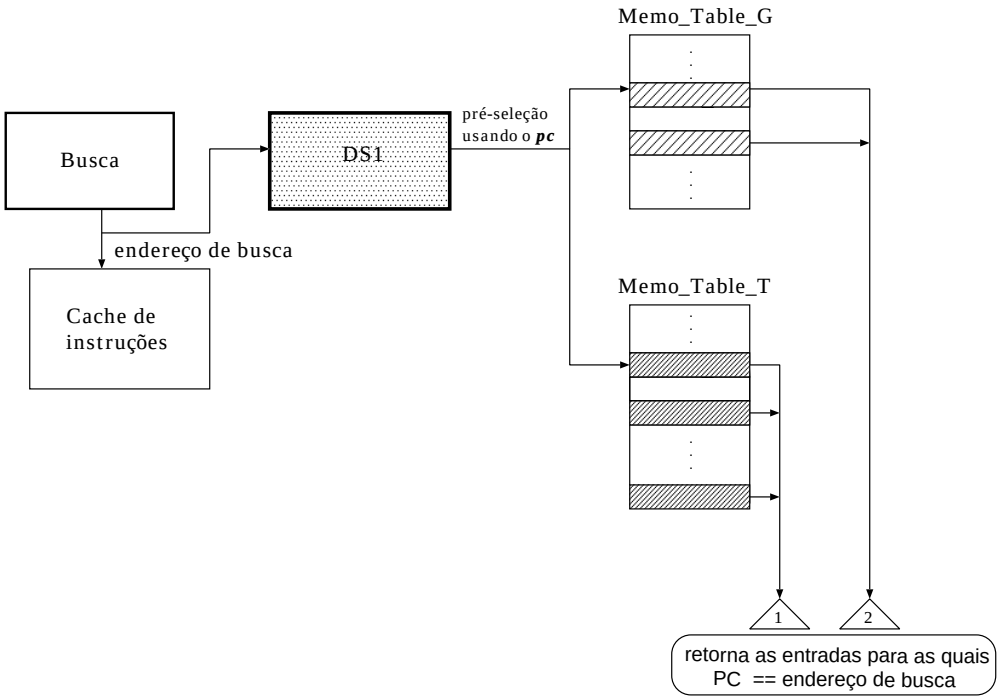


Figura 4.3: Microarquitetura do ciclo de busca.

A Figura 4.4 apresenta os passos efetuados pelo estágio **DS2**, onde são identificadas instruções ou traces redundantes. Nesta etapa, as tabelas de memorização

são interrogadas considerando as entradas candidatas que foram pré-selecionadas. Os passos são descritos pelas operações **Op(3)** e **Op(6)**.

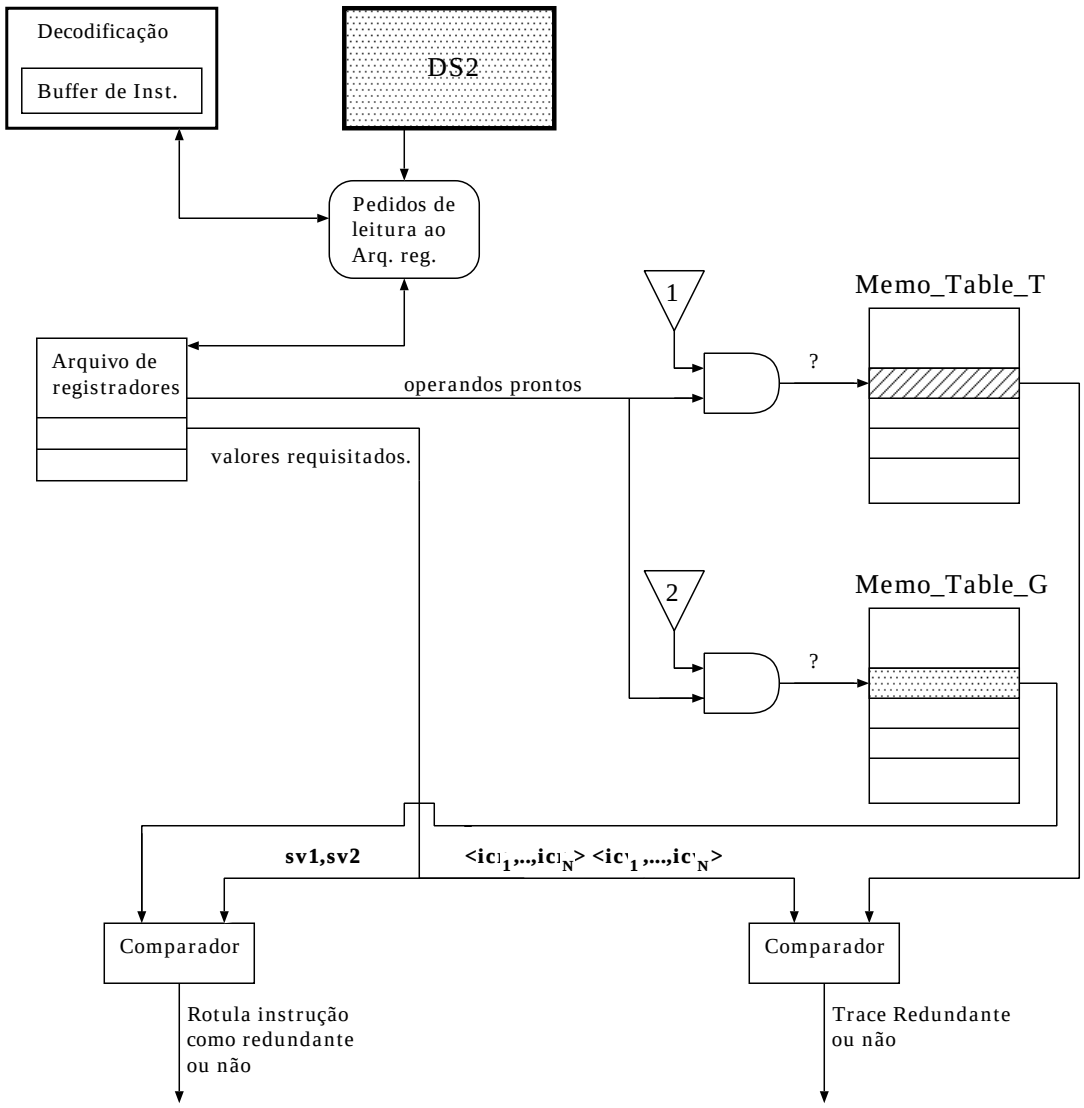


Figura 4.4: Microarquitetura do ciclo de decodificação.

As Figuras 4.5 e 4.6, apresentam os passos efetuados pelos estágios **DS2** e **DS3**, quando da ocorrência ou não de instruções ou traces redundantes, bem como a construção de traces. Os passos são descritos pelas operações **Op(4)** e **Op(7)** anteriormente estabelecidas.

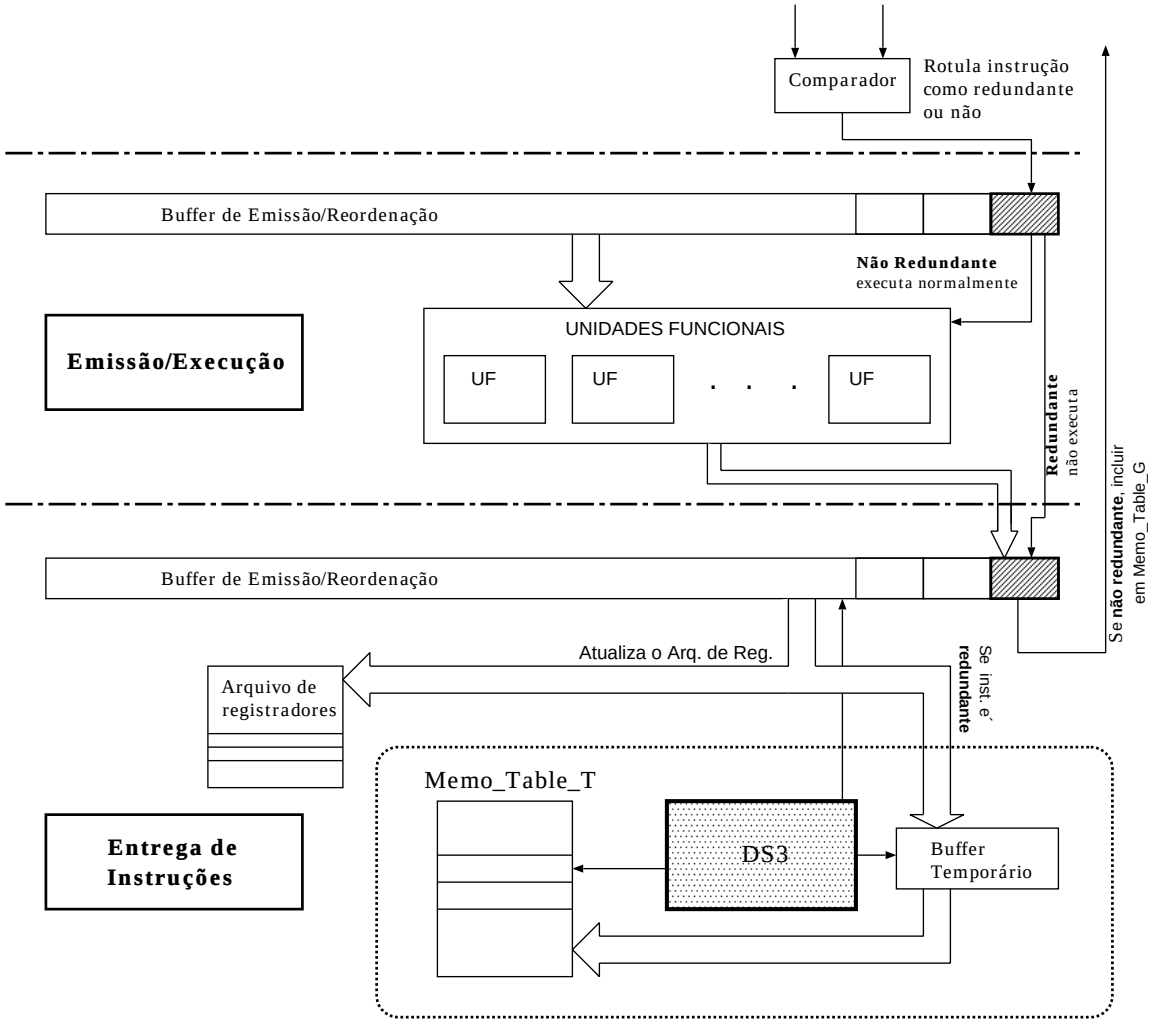


Figura 4.5: Microarquitetura reusando instruções simples no *DTM*.

4.3 Algumas considerações relativas à adição do mecanismo *DTM*

Esta seção pretende preencher algumas lacunas quanto a interação entre o mecanismo *DTM* e a microarquitetura superescalar substrato. Intencionalmente, foram deixadas para esta seção, algumas considerações não explícitamente mencionadas anteriormente, de forma a serem cobertas por exemplos, figuras e uma descrição mais dependente do modelo de execução superescalar. As subseções a seguir serão direcionadas a problemas particulares e mais detalhadamente discutidos.

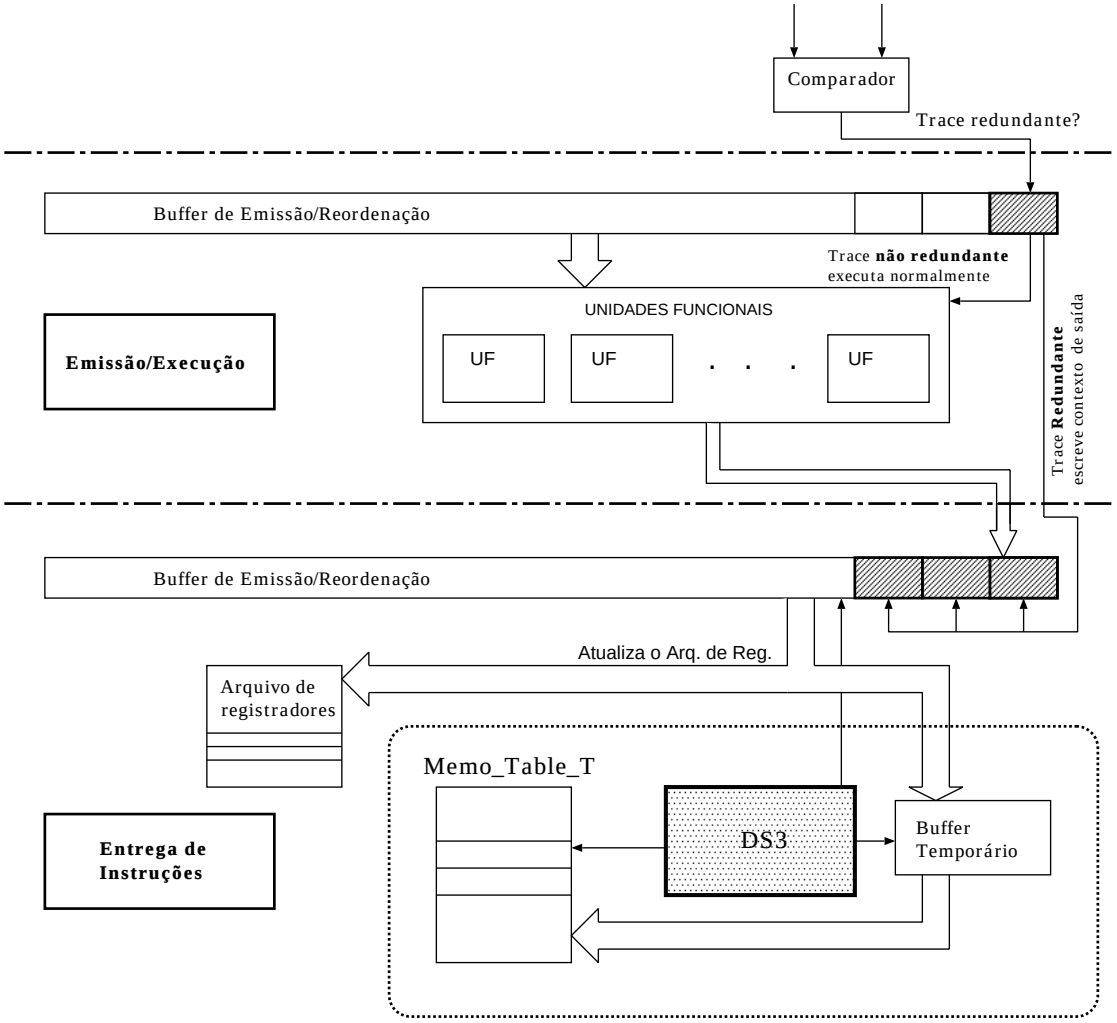


Figura 4.6: Microarquitetura reusando traces no *DTM*.

4.3.1 Considerações sobre a construção de traces

No estágio **DS3**, a construção de um novo trace pode iniciar-se no mesmo ciclo de processador no qual um trace anteriormente construído foi finalizado (dado que exista uma instrução não redundante dentre as instruções que foram analisadas no mesmo ciclo). Usando-se alternativamente buffers temporários, o *DTM* pode iniciar a construção de um novo trace, antes do contexto de um trace anteriormente construído ser salvo. Por exemplo, se o *DTM* está usando correntemente o *Buffer_T#1* para a construção de um trace Tn, então ele poderá alocar o *Buffer_T#2* para a construção de um novo trace $Tn + 1$ (iniciando-se provavelmente no mesmo ciclo), enquanto o contexto é salvo em *Buffer_T#1*. O buffer *Buffer_T#1* pode então alternativamente, ser alocado para um próximo trace $Tn + 2$, se a mesma situação

inicialmente descrita ocorrer novamente. Dado que a construção de um trace tenha terminado, o *DTM* transfere os contextos montados nos buffers temporários para a tabela de memorização de traces *Memo_Table_T*. A composição dos buffers temporários é idêntica a composição de uma entrada de *Memo_Table_T*.

A Figura 4.7, apresenta o formato dos buffers temporários *Buffer_T#1* e *Buffer_T#2*, e a configuração de seleção necessária para permitir a construção de dois traces em um mesmo ciclo de clock.

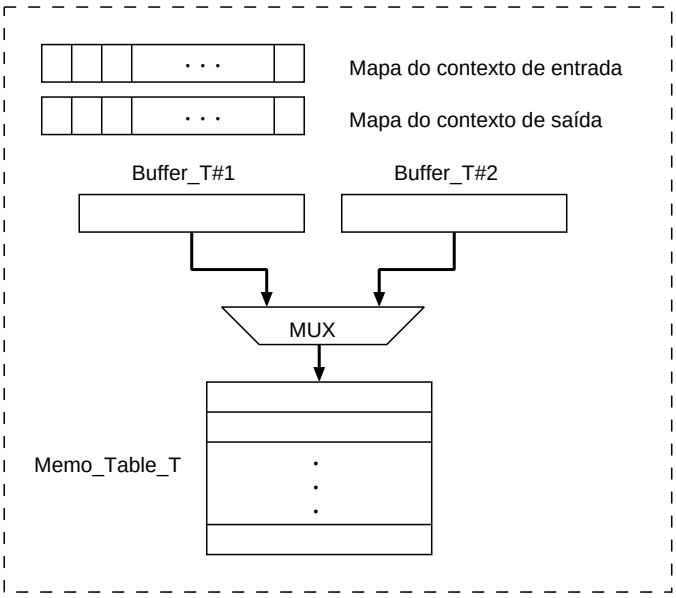


Figura 4.7: Unidade para construção de traces.

Uma outra consideração com relação a construção de traces, faz referência a identificação e rotulação de instruções como sendo redundantes ou não redundantes. Como já mencionado anteriormente (quando da adição do mecanismo *DTM* em uma arquitetura substrato superescalar), uma instrução será redundante se possuir todos os seus operandos instanciados por valores válidos quando da efetuação do teste de reuso, e esta instrução instanciada estiver presente na tabela de memorização. Considerando as restrições impostas pelo modelo de execução superescalar, observou-se que nem todas as instruções redundantes são passíveis de reuso.

A Figura 4.8(a), apresenta um exemplo de uma seqüência de instruções dinâmicas que serão avaliadas quanto a sua redundância. Pode-se observar na Figura 4.8(b), que as instruções *100* e *104* possuem todos os operandos de entrada instanciados com valores válidos, logo estas instruções serão comparadas com as respectivas entradas

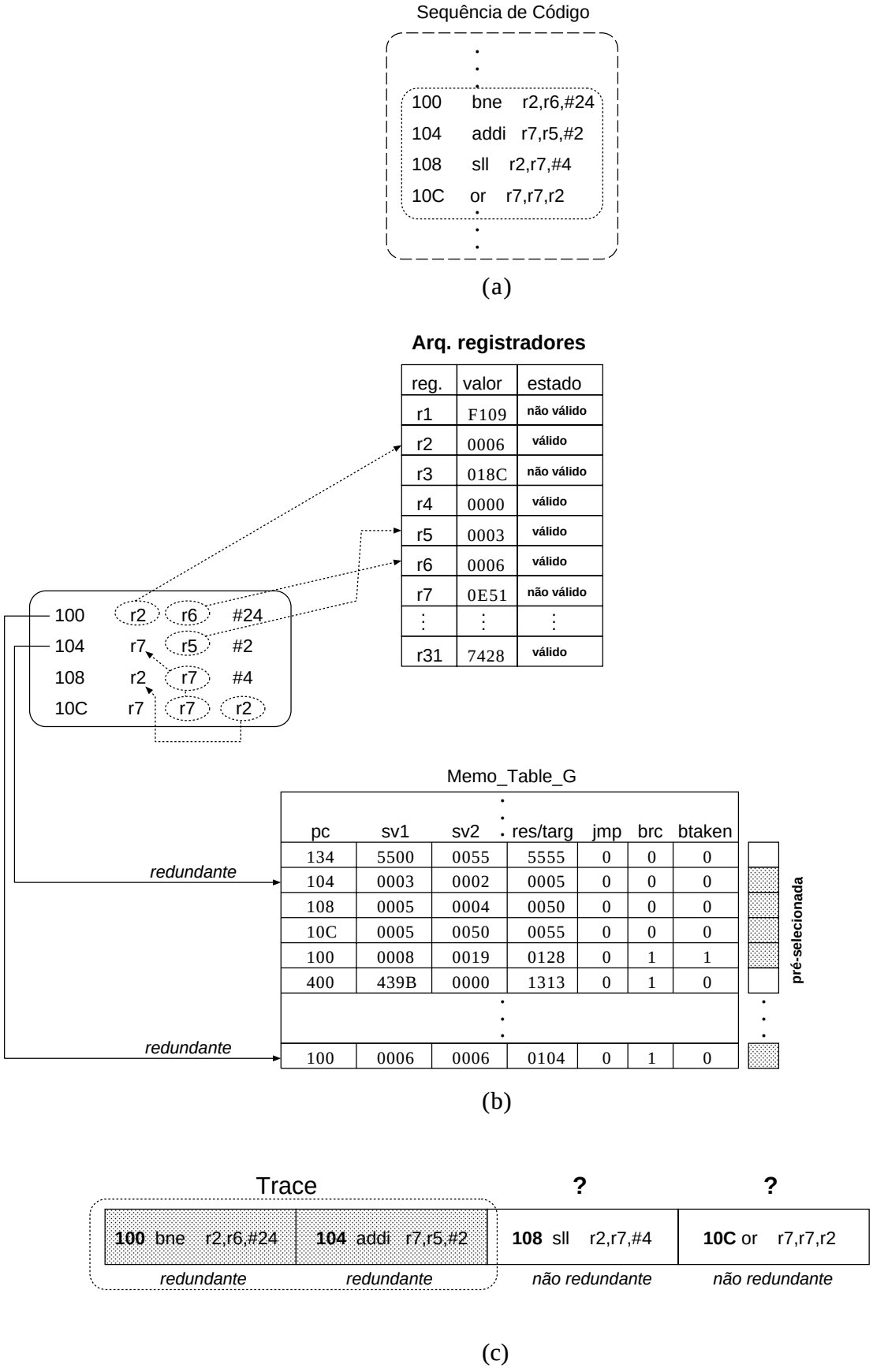


Figura 4.8: Construindo traces somente com instruções redundantes.

pré-selecionadas em *Memo_Table_G* (no estágio anterior). Visto que existem entradas em *Memo_Table_G* que correspondem as instâncias das instruções *100* e *104*, estas instruções serão marcadas como redundantes. O mesmo não acontece com as instruções *108* e *10C*, visto que estas não possuem seus operandos de entrada instanciados com valores válidos. Para o exemplo observado, a instrução *108* depende do resultado de *r7* que é produzido por *104*, e a instrução *10C* depende dos resultados de *r7* e *r2* produzidos por *104* e *108* respectivamente. Convém notar, que apesar do registrador *r2* estar marcado como possuindo um valor válido armazenado, esta marcação só é válida para a instrução *100* e não é válido para *10C*, pois *r2* será produzido por uma instrução entre *100* e *10C* (instrução *108* no caso).

Avaliando ainda o exemplo apresentado na Figura 4.8(c), o trace a ser construído incluirá somente as instruções *100* e *104*, pois estas estão marcadas como redundantes, enquanto *108* e *10C* estarão marcadas como não redundantes. Observando-se o trace construído, verifica-se que as instruções *108* e *10C*, apesar de não serem redundantes pelo procedimento descrito, são decorrentes da redundância de suas entradas dependentes dos resultados produzidos pelas instruções *100* e *104*, logo serão redundantes também, e deverão ser inclusas no trace construído.

A partir desta constatação foi proposto um procedimento para remarcação dinâmica dos rótulos atribuídos às instruções marcadas como não redundantes. A remarcação poderá ser efetuada por um dos três procedimentos propostos abaixo (considerando apenas as instruções marcadas como não redundantes e pertencentes ao domínio de instruções válidas do *DTM*):

(i) Quando uma instrução rotulada como não redundante possuir todos os seus operandos instanciados para execução, esta será pesquisada em *Memo_Table_G*. Sua instância corrente será remarcada pelo resultado da pesquisa, se estiver presente em *Memo_Table_G* será remarcada como redundante, caso contrário continuará marcada como não redundante;

(ii) Quando uma instrução rotulada como não redundante receber os valores de seus operandos de entrada a partir de instruções rotuladas como redundantes, esta será remarcada como redundante;

(iii) A conjunção de (i) e (ii).

A solução (i), apesar de funcionar corretamente, poderá remarcar como redundante, instruções que freqüentemente não possuem seus operandos de entrada prontos durante o teste de reuso, e estas poderão ser inclusas nos traces em formação. Esta situação pode produzir traces que raramente serão reusados. Além desta consideração, a implementação desta solução pode aumentar (não muito significativamente) o número de portas de acesso à *Memo_Table_G*. Nenhum problema de temporização será adicionado ao se escolher esta solução, visto que o acesso à *Memo_Table_G* neste caso, irá ocorrer paralelamente à execução da instrução. Quando uma instrução não redundante for completamente instanciada, esta será executada e posteriormente repassada para o *Estágio de Entrega*, decorrendo então dois ciclos de clock (o mesmo tempo estipulado para um acesso em dois estágios à *Memo_Table_G*). Para este caso, a instrução a ser remarcada não é reusada.

A solução (ii) é implementada mais facilmente e decorre da própria descrição de funcionamento da arquitetura substrato feita na seção 4.1. Uma vez que uma instrução tenha sido processada pela unidade funcional, o *Estágio de Execução* reenvia o resultado da instrução para o *Estágio de Emissão*. Este por sua vez, aloca o resultado recebido para as instruções que o utilizarão como entrada. Explorando esta funcionalidade, basta incluir em cada resultado produzido pela unidade funcional, um bit indicando se o resultado foi produzido por uma instrução marcada como redundante ou não redundante. Este bit será responsável pela remarcação da instrução que o recebe. Esta solução exige a inclusão de mais uma linha nos barramentos de *forwarding* para carregar a informação de remarcação. Um caso especial merece atenção, este ocorre quando uma instrução com dois operandos de entrada e avaliada para remarcação, recebe um dos valores como produzido por uma instrução redundante e o outro valor como produzido por uma instrução não redundante.

A solução (iii) considera a implementação adotada em (ii), e trata seu caso especial por intermédio de (i). Esta solução alivia os problemas de (i) e (ii). A Figura 4.9, apresenta um exemplo onde a solução (iii) é implementada.

A Figura 4.9(a) apresenta a seqüência de código que será avaliada, enquanto a Figura 4.9(b) apresenta a rotulação de instruções como redundantes ou não redundantes de acordo com o procedimento determinado pela condição (iii). Para a instrução *100*, seu operando de entrada *r6* possui um dado válido no momento da

efetuação do teste de reuso, logo a instrução encontra-se completamente instanciada e uma pesquisa em *Memo_Table_G* será efetuada para verificar a redundância desta. Considerando o exemplo, uma busca em *Memo_Table_G* identificou a instrução *100* como presente, logo esta é marcada como redundante e conseqüentemente é reusada. As demais instruções são marcadas como não redundantes, pois não possuem seus operandos prontos no momento do teste de reuso (não instanciadas), estas são: a instrução *104* depende do valor de *r5* produzido pela instrução *100*; a instrução *108* depende do valor de *r7* produzido pela instrução *104* e *r4* que possui um dado inválido; a instrução *10C* depende do valor de *r7* e *r2* produzidos pelas instruções *104* e *108* respectivamente; a instrução *110* depende do valor de *r2* produzido por *108* e *r8* que possui um dado inválido.

A Figura 4.9(b) apresenta as instruções despachadas para o *Buffer de Emissão* e aguardando execução. A instrução *100* não será executada pois é redundante e portanto reusada. Quando o resultado da instrução *100* (marcado como redundante) for reenviado para o *Buffer de Emissão*, pela condição (ii), a instrução *104* será marcada como redundante. Após a execução da instrução *104*, será reenviado o seu resultado (marcado como redundante) para o operando *r7* da instrução *108*, configurando deste modo a condição (ii). Entretanto, o outro operando (*r4*) da instrução *108* continua inválido e não é produzido por nenhuma instrução redundante. A partir desta constatação, a condição (i) deverá ser avaliada, ou seja, quando *r4* possuir um dado válido, a instrução *108* estará completamente instanciada e será pesquisada em *Memo_Table_G*. Para o exemplo apresentado, a instrução *108* instanciada, encontra-se em *Memo_Table_G* e logo será marcada como redundante. A instrução *10C* irá receber em seus operandos de entrada, registradores *r7* e *r2*, os valores resultantes da execução das instruções *104* e *108* respectivamente (marcados como redundantes). Logo, pela condição (ii) a instrução *10C* será marcada como redundante. A instrução *110* irá receber em um de seus operandos de entrada (o registrador *r2*) o valor resultante da execução da instrução *108* (marcado como redundante). Entretanto o outro operando de entrada, o registrador *r8*, não é produzido por nenhuma instrução redundante. A partir do exposto, a remarcação da instrução *110* depende do resultado avaliado pela condição (i). Quando *r8* possuir um dado válido, a instrução *110* estará completamente instanciada e será buscada

em *Memo_Table_G*. Para o exemplo apresentado, a instrução *110* instanciada, não encontra-se em *Memo_Table_G* e logo continuará marcada como não redundante.

As considerações apresentadas decorrem da heurística definida para a seleção de instruções que serão usadas para a construção de traces (traces serão compostos por instruções identificadas como redundantes). Para propósitos de avaliação, a solução (i) será escolhida para implementação neste trabalho, mesmo considerando que esta solução possibilita a inserção de instruções redundantes que podem não possuir seus operandos de entrada prontos quando do teste de reuso, impedindo assim o reuso do trace construído. Esta situação apesar de inibir parcialmente o reuso de traces construídos, permite uma implementação de preditores de valores [29], aptos a predizer os valores de entrada não instanciados aos traces (estendendo a predição de valores aplicada a instruções). É importante ressaltar que nenhuma destas soluções propostas provocam atrasos a quaisquer elementos do pipeline que é implementado na arquitetura substrato, visto que as operações relacionadas às soluções propostas são externas ao caminho crítico de execução.

4.3.2 Tratando instruções de desvio no *DTM*

Como mencionado anteriormente durante a descrição do mecanismo *DTM*, o estado do preditor de desvios deverá ser atualizado sempre que uma instrução de desvio for reusada (ou individualmente ou como parte de um trace). Convém mencionar que o preditor de desvios (independente do tipo) será atualizado com relação ao seu estado de predição, somente no *Estágio de Entrega*.

Em alguns preditores de desvios [59, 47], a atualização do estado de predição envolve a seleção de uma entrada da tabela de predição, usando-se para este fim, um tag ou índice determinado pelo endereço de memória da instrução de desvio. Considerando estes preditores, quando uma instrução de desvio é reusada isoladamente, seu endereço estará disponível no campo *pc* de sua correspondente entrada em *Memo_Table_G*. A entrada do preditor de desvios que contém o estado de uma predição selecionada é então atualizada com o resultado do desvio indicado pelo campo *btaken*. Entretanto, para instruções de desvio que são reusadas como parte de um trace, o mecanismo *DTM* como descrito anteriormente, poderia ser adaptado para acomodar o endereço da instrução de desvio. Possíveis soluções:

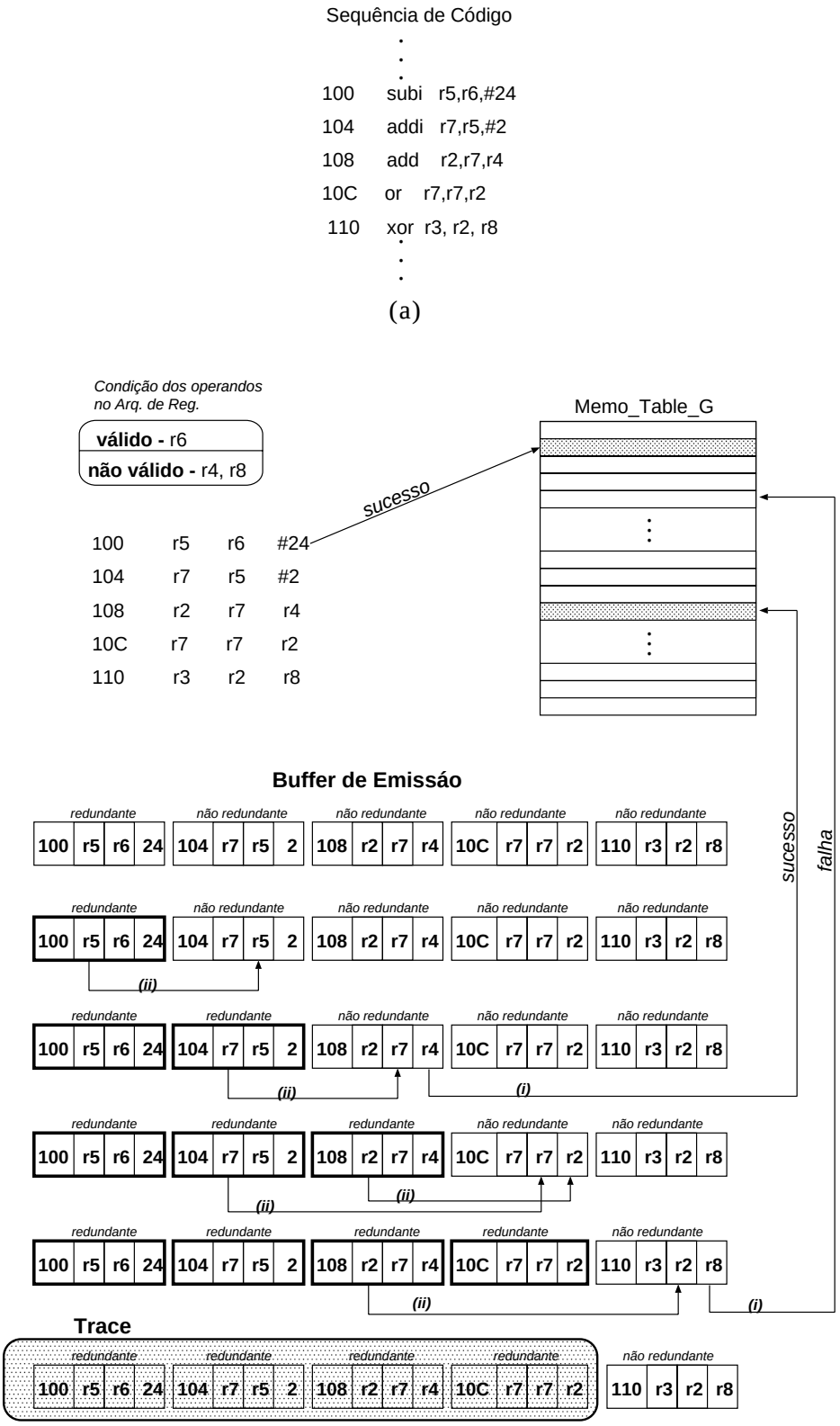


Figura 4.9: Remarcação dinâmica de instruções não redundantes, (a) seqüência a ser avaliada, (b) procedimento de rotulação.

(i) Novos campos seriam adicionados para cada entrada de *Memo_Table_T*, estes campos seriam inseridos com a função de armazenar os endereços de memória das instruções de desvio (ou alguns bits destas) pertencentes ao trace. Entretanto, esta solução pode aumentar o custo do mecanismo *DTM* (dependendo do número de entradas de *Memo_Table_T*);

(ii) Cada campo do contexto de entrada/saída (*icv*/*ocv*) livre (não alocado), poderia ser utilizado para armazenar o endereço de memória de uma instrução de desvio (ou parte desta). Em decorrência do compartilhamento do campo *icv*/*ocv* entre entrada/resultado e endereços, o mecanismo *DTM* modificado pode requerer um contexto de entrada/saída com um maior número de campos (N), de modo a preservar o mesmo potencial de reuso do mecanismo *DTM* que não compartilha os campos do contexto de entrada/saída. Entretanto, o custo adicional introduzido por esta alternativa é ainda menor que o custo decorrente da solução (i).

Em se considerando preditores de desvios que não necessitem da informação de endereço de memória da instrução de desvio [70], o mecanismo *DTM* como descrito originalmente, seria capaz de suportar a atualização dos estados de predição sem qualquer modificação e através dos campos *bmask* e *btaken*. São exemplos destes preditores, o *preditor de desvio com dois níveis de história e padrão de história global* [70]. Estes se caracterizam por serem preditores de desvios por correlação, e podem aumentar a precisão da predição, pois o resultado de um desvio tende a ser correlacionado aos resultados de desvios anteriores. O preditor de desvio por correlação utiliza um registrador de história de desvios *BHR* e uma tabela de história de padrões *PHT* como apresentado na Figura 4.10. O *BHR* é um registrador de deslocamento que armazena o resultado dos últimos desvios executados (0-não tomado e 1- tomado), e é atualizado cada vez que um desvio é executado, sendo o seu resultado inserido através do deslocamento dos desvios anteriores. Um *BHR* possuindo k bits, guarda o resultado de história dos últimos k desvios executados. Em um preditor por correlação global, existe um único *BHR* que é atualizado por todos os desvios. A *PHT* é uma tabela que possui 2^k entrada e armazena em cada entrada um contador de dois bits. Para predizer o próximo desvio, o *BHR* é usado para indexar a *PHT* e a predição é efetuada considerando-se o valor armazenado

nesta entrada. Em resumo, a predição corrente depende de:

(1) O padrão dos k desvios precedentes em *BHR*;

(2) O comportamento do desvio nas últimas vezes em que o padrão foi encontrado (contador de 2 bits na *PHT*).

Quando o resultado de um desvio é conhecido, ele é usado para atualizar o mesmo contador de 2 bits da *PHT* (incrementa se tomado, decrementa se não tomado).

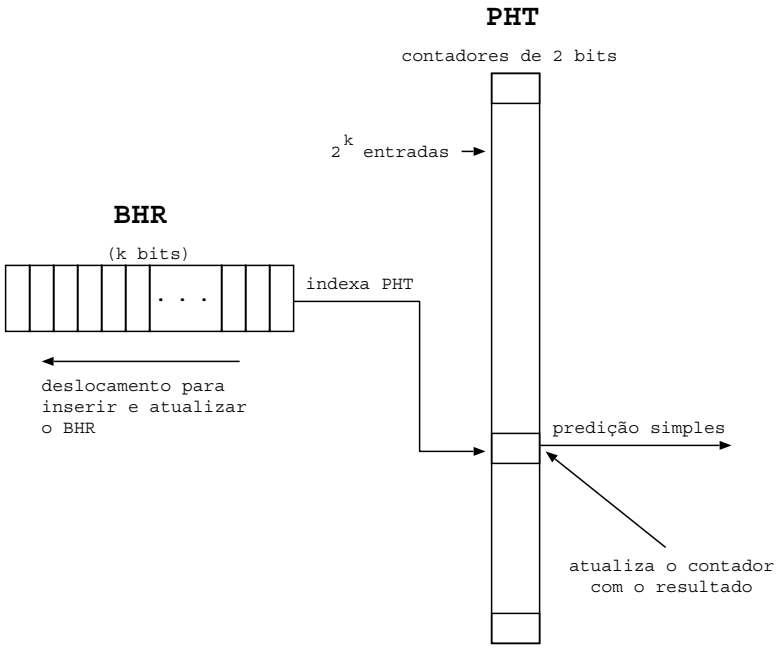


Figura 4.10: Preditor de desvio por correlação com dois níveis de história e padrão de história global.

A partir do exposto, torna-se razoável afirmar que o custo do mecanismo *DTM* é mais favorecido em microarquiteturas empregando preditores de desvio com dois níveis de história e padrão de história global. Isto não constitui qualquer desvantagem, pois como apresentado em [70], sob um custo de implementação constante, a precisão da predição obtida pelo esquema de história global é ligeiramente inferior à apresentada por esquemas por endereços ou por conjuntos. Este tipo de preditor têm sido satisfatoriamente aplicado em trabalhos recentes [56, 39].

4.3.3 Redirecionando o fluxo de execução após o reuso de um trace redundante

Como descrito anteriormente, quando um trace é reusado o contador de programa *PC* será carregado com o endereço da instrução subseqüente a última instrução do trace reusado (ou seja, o *npc* da última instrução). Dependendo do modelo arquitetural considerado (uma arquitetura superescalar neste caso), o redirecionamento do fluxo de instrução decorrente do reuso de um trace pode acarretar indesejáveis descontinuidades de execução. A Figura 4.11 apresenta um exemplo (considerando somente os estágios de busca e decodificação/despacho de instruções), onde o redirecionamento de execução proporcionará efeitos negativos. Neste exemplo, o trace identificado como redundante é composto pelas instruções *100*, *104* e *108*. Este trace ao ser reusado atribuirá ao *PC* o endereço *10C* correspondente a próxima instrução a ser executada após o reuso do trace, e este endereço será referenciado como o próximo endereço a ser buscado no cache de instruções. Observando que um acesso ao cache de instruções está em curso e referenciando a linha que contém o endereço *110*, um redirecionamento efetuado pelo trace para o endereço *10C* e desconsideração das instruções que estão sendo buscadas, indesejávelmente irá proporcionar um retrocesso no histórico da execução.

A partir do exposto, torna-se necessário ser seletivo quanto a atitude de redirecionamento do fluxo de execução quando um trace for reusado. Alguns critérios foram adotados, de modo a eliminar os efeitos negativos. Um trace redundante irá redirecionar o fluxo de execução nas seguintes situações:

(1) Se forem identificadas predições incorretas;

(2) Se o trace encapsular um maior número de predições do que as expostas pelo preditor;

(3) Se as predições identificadas forem corretas, porém a última predição identificada encontra-se no estágio de busca e esta é predita como tomada.

A Figura 4.12 esboça a organização estrutural necessária para identificar as ocorrências expostas por (1), (2) e (3). Nesta, é apresentada a configuração dos

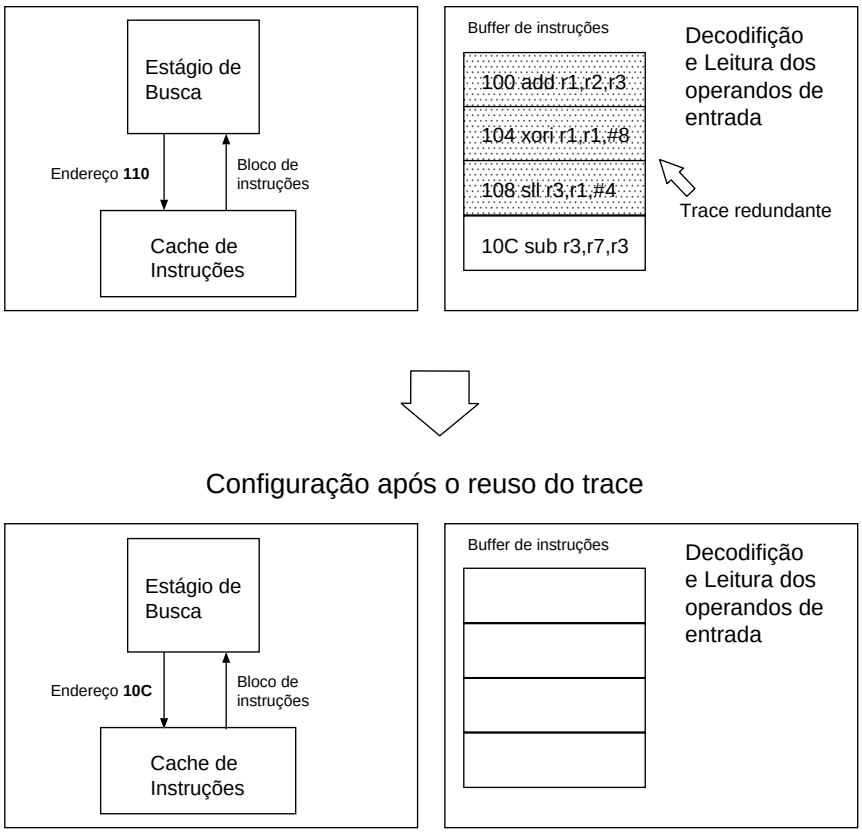


Figura 4.11: Efeito negativo decorrente do redirecionamento do fluxo de execução imposto pelo reuso de um trace.

estágios de busca e decodificação durante a execução normal de um programa. A representação distingue:

• O buffer de instruções preenchido com instruções fornecidas pelo estágio de busca. Neste também são explicitados separadamente os campos *dmask* e *dtaken*, que indicam respectivamente quais instruções são desvios e qual a predição efetuada;

• O estágio de busca, onde é efetuada a busca das próximas instruções a serem executadas (em paralelo ao estágio de decodificação), a predição das mesmas, e são preenchidos os campos *fmask* e *ftaken* que indicam respectivamente, quais instruções são desvios e qual a predição efetuada. Estas informações serão transferidas para o estágio de decodificação no próximo ciclo, ou seja, *dmask* e *dtaken* receberão os valores de *fmask* e *ftaken* respectivamente.

Quando um trace redundante é identificado para reuso, verifica-se a possibilidade de redirecionamento do fluxo de execução, através da verificação dos desvios

marcados e preditos em *dmask*, *dtaken*, *fmask* e *ftaken*. Nesta ordem, estes campos serão comparados aos respectivos campos *bmask* e *btaken*, armazenados na entrada de *Memo_Table_T* correspondente ao trace identificado como redundante.

Na Figura 4.12 é esboçada a seqüência de instruções a serem despachadas para execução (presentes no buffer de instruções), seguida da seqüência de instruções sendo acessadas no cache de instruções e preditas para posterior envio para o buffer de instruções (decodificação e despacho). Esta seqüência de instruções é determinada pelas instruções de desvio identificadas e preditas. Observando-se a configuração destes estágios no processador, identifica-se:

- A instrução *100* como a instrução inicial (no estágio de decodificação);
- A instrução *11C* como a instrução final (no estágio de busca);
- O endereço *120* como o endereço da próxima instrução a ser acessada no cache de instruções (no próximo ciclo, pois é subseqüente à instrução 11C);
- As instruções de desvio *104* e *118* preditas como não tomadas.

A Figura 4.12 considera a existência de um trace redundante identificado com *pc=104* e *npc = 418*, possuindo as instruções de desvio *104* e *118* marcadas como não tomada e tomada respectivamente. O trace em questão, encapsula as instruções *104*, *108*, *10C*, *110*, *114* e *118*. A Figura 4.13 apresenta exemplos de como as condições expostas para redirecionamento do fluxo de execução são identificadas. Como apresentado no exemplo da Figura 4.13(a), esta caracteriza a condição (1), pois observa-se a ocorrência de uma predição incorreta feita no estágio de busca e identificada nos campos *fmask* e *ftaken*. Neste caso, as marcações são diferentes das marcações armazenadas nos campos *bmask* e *btaken* (em particular no campo *ftaken* e *btaken*). A partir do mesmo exemplo, na Figura 4.13(b) justifica-se (2), pois é imediato notar que existem predições no trace redundante que não são identificadas em *dmask* ou *fmask*. A condição (3) pode ser justificada pelo exemplo da Figura 4.13(c), onde neste caso, procura-se tirar proveito de um desvio predito como tomado e identificado como correto. Considerando que o fluxo será redirecionado por este desvio e o desvio encontra-se encapsulado no trace redundante a ser reusado, não ocorrerá penalidade alguma ao se redirecionar a busca da próxima instrução a ser executada após o reuso do trace.

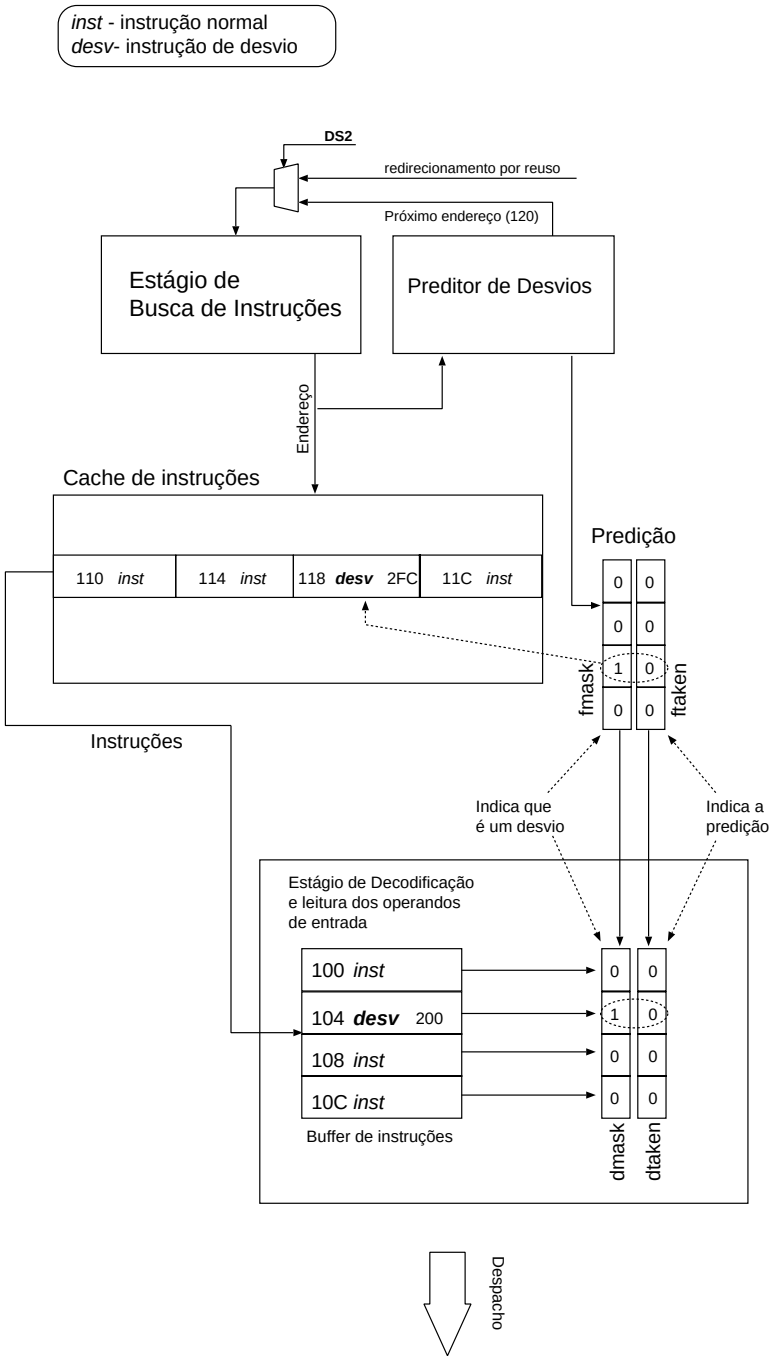
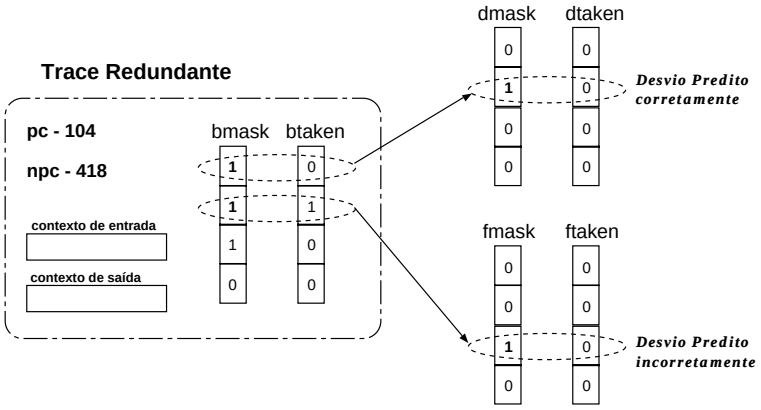
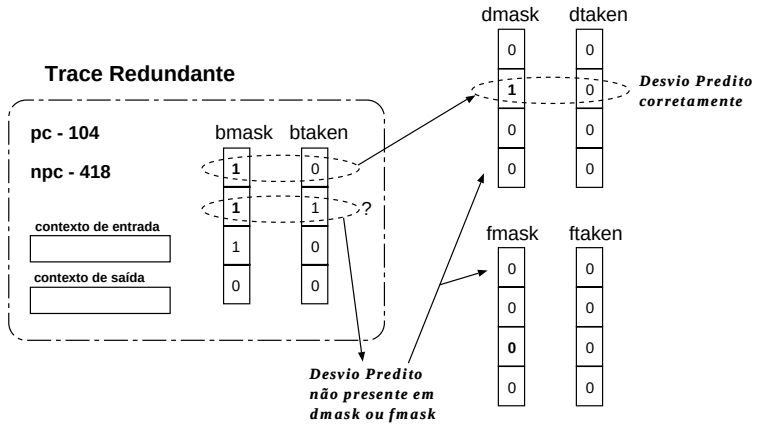


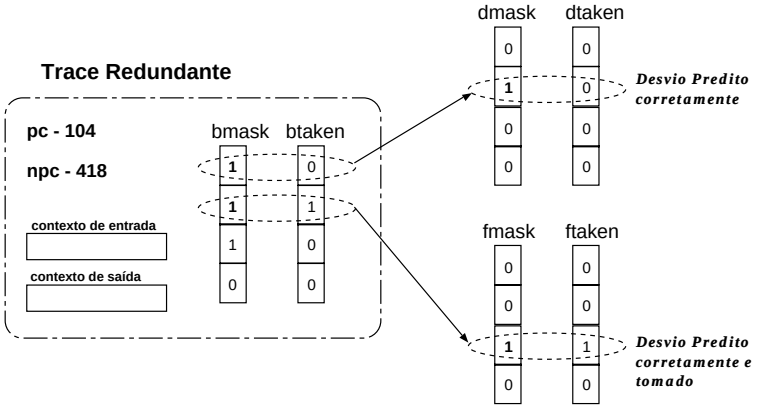
Figura 4.12: Estágio de busca e decodificação preenchidos, identificação dos desvios e suas predições.



(a)



(b)



(c)

Figura 4.13: Exemplos das situações consideradas para o redirecionamento da busca de instruções.

Outras considerações para redirecionamento do fluxo de execução, atentam para os acessos sem sucesso e requisitados ao cache de instruções, e a impossibilidade de se despachar instruções para o buffer de emissão quando este apresenta-se completamente preenchido.

As considerações descritas anteriormente descrevem em que condições o fluxo de execução pode ser redirecionado. Entretanto, deve-se considerar ainda, que as instruções encapsuladas em um trace redundante e já presentes no buffer de instruções e/ou sendo buscadas no cache de instruções, devem ser desconsideradas para o despacho. Na Figura 4.14 é apresentado um trace redundante incluindo um novo campo denominado *i-mask*. A inclusão deste campo possui como objetivo: preencher uma máscara de inibição incluída no processador, de modo a evitar que as instruções do trace redundante que se encontram presentes no buffer de instruções e as instruções que estão sendo buscadas pelo estágio de busca, sejam alocadas em alguma entrada do buffer de emissão. Quando um trace redundante for identificado para reuso, seu campo *i-mask* será carregado na máscara de inibição *mask_inib* (adicionada ao processador substrato) nas correspondentes posições (entre o buffer de instruções e o despacho), a partir da primeira instrução do trace redundante. Como apresentado na Figura 4.14, o valor *1* em uma posição de *mask_inib* inibe a alocação da correspondente instrução do buffer de instruções para o buffer de emissão, enquanto *0* habilita a alocação da correspondente instrução. A máscara de inibição *mask_inib* anterior ao buffer instruções, também será preenchida pelo *i-mask*, com o objetivo de desabilitar uma posterior decodificação de instruções que pertencem ao trace e que serão enviadas no próximo ciclo (as instruções desabilitadas, não chegarão nem mesmo a serem carregadas no buffer de instruções).

Para o exemplo apresentado na Figura 4.14, foi considerado como identificado um trace redundante, estando este, apto para reuso. O trace em questão, encapsula o trecho de código dinâmico delimitado pelos endereços *104* a *118* e armazena o endereço *11C* correspondente ao endereço da próxima instrução a ser executada após o reuso do trace. Pode-se observar através da configuração do campo *i-mask*, que as instruções do trace serão desconsideradas para o despacho (em seu lugar, será despachado o contexto de saída do trace). Para este caso, a instrução no endereço *11C* será normalmente passada para o estágio de decodificação e a busca

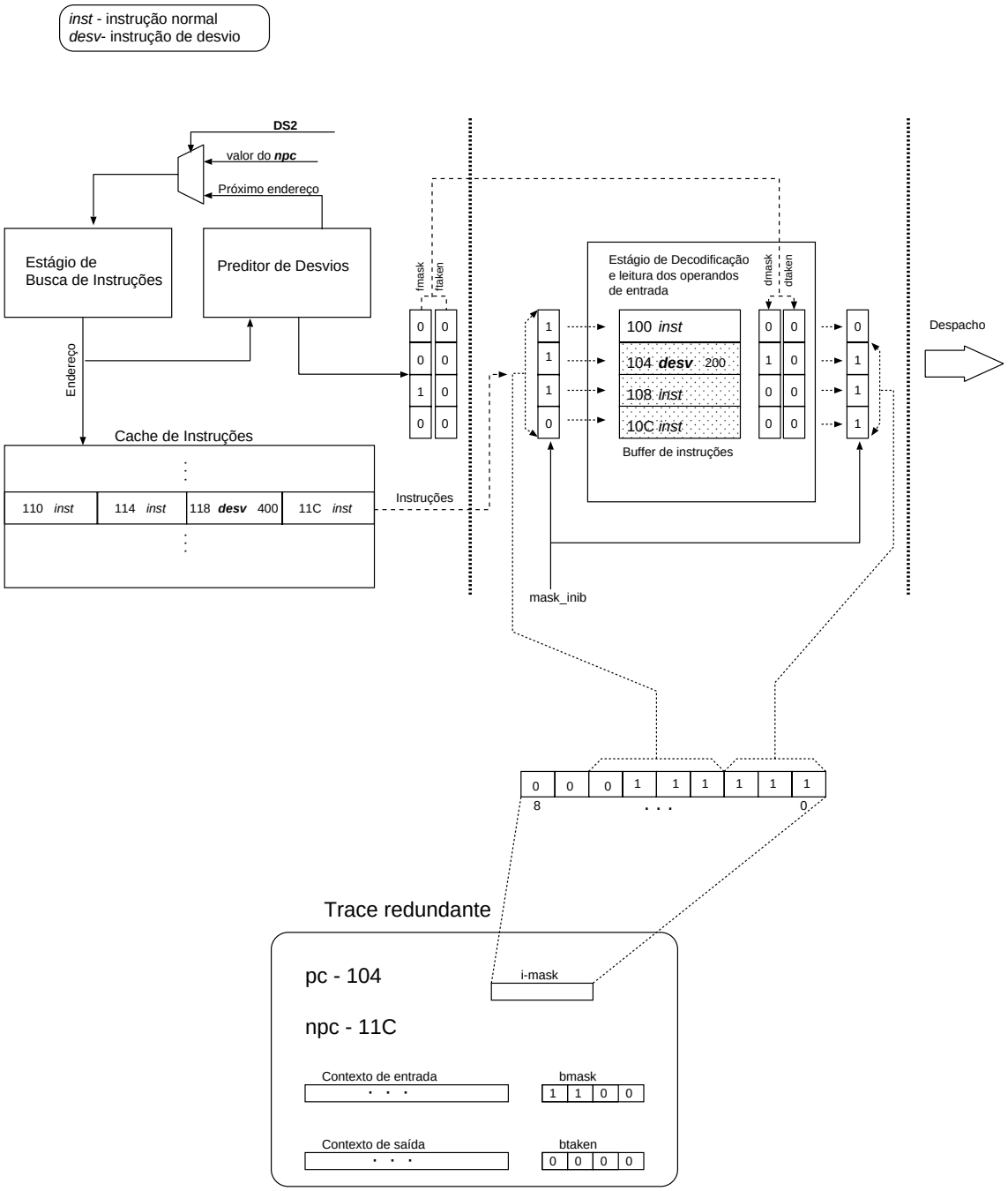


Figura 4.14: Inibindo o despacho de instruções encapsuladas em um trace redundante.

das próximas instruções continuará normalmente. Vale ressaltar, que a máscara de inibição *i-mask* cobrirá apenas as instruções que estão nos estágios de decodificação e busca (nesta ordem). Se existir pelo menos um bit da máscara *i-mask* que esteja em *1* e não escalonado para *mask_inib*, este bit, determina que o fluxo de execução seja redirecionado via o campo *npc* do trace redundante.

O tamanho do campo *i-mask* é parâmetro definido em função da largura de busca e decodificação/despacho do processador substrato (4 instruções por ciclo). Considerando wb a largura de busca, ou seja, o número máximo de instruções enviadas por ciclo do estágio de busca para o estágio de decodificação; wd a largura de decodificação/despacho, ou seja, o número máximo de instruções por ciclo que podem ser decodificadas e despachadas; s o número da entrada do buffer de instruções que armazena a primeira instrução do trace redundante (0,1,2,3); e l o número de instruções no trace redundante. A Figura 4.15, apresenta um exemplo onde $wb = 4$, $wd = 4$, $s = 1$ e $l = 10$ (o trace em questão possui 10 instruções delimitadas pelos endereços *104* à *128*). Para este exemplo, o bit **7** de *i-mask* possui o valor *1* e não foi escalonado para *mask_inib*. Esta configuração indica que o trace redundante não está inserido completamente no processador, logo o fluxo de execução deverá ser redirecionado pelo valor de *npc*, e as instruções atualmente presentes no buffer de instruções e as acessadas pelo estágio de busca, serão desconsideradas para despacho.

A justificativa para o número de elementos de *i-mask* (igual a 9 no exemplo), considera que este é o valor extremo, e decorre de $wb - s + wd + 1$, considerando o caso em que a primeira instrução do trace redundante ocupe a primeira entrada do buffer de instruções ($s = 0$) e $l > wb + wd$. O oitavo bit para este caso, será utilizado para efetuar o redirecionamento do fluxo de execução, visto que todos os 8 bits de *i-mask* foram escalonados em *mask_inib* e $l > wb + wd$.

4.3.4 Considerações sobre a ativação dos estágios do *DTM*

Considerando os estágios do mecanismo *DTM*, **DS1**, **DS2** e **DS3**, estes operam em paralelo com os estágios de busca, decodificação e entrega respectivamente e, é esperado não adicionarem quaisquer níveis de profundidade ao pipeline. O estágio **DS1** executa principalmente, buscas associativas nas tabelas de memorização, e deste modo, requer um ciclo de tempo comparável ao utilizado pelo cache de instruções.

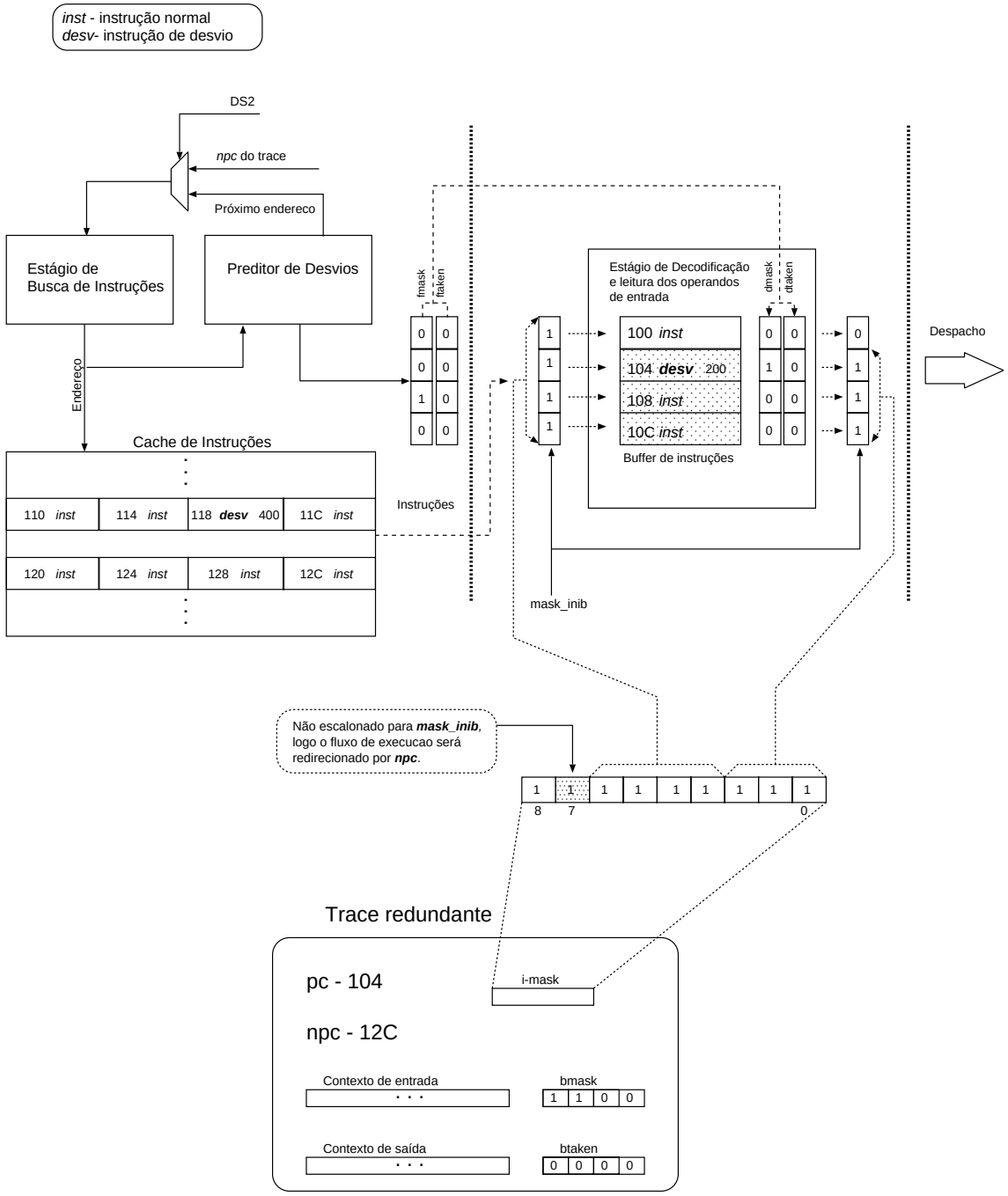


Figura 4.15: Exemplo de inibição de instruções encapsuladas em um trace redundante e redirecionamento do fluxo de execução.

Isto significa que o estágio **DS1** do *DTM* não adiciona tempo de ciclo algum, pois sua operação pode ser completamente sobreposta pelo acesso ao cache de instruções.

As operações do estágio **DS2** do *DTM* também são transparentes. O estágio **DS2** pode ler os valores do contexto de entrada (para as entradas pré-selecionadas pelo estágio **DS1**) em conjunto com a requisição feita pelo *Estágio de Decodificação* para leitura dos operandos das instruções. A pesquisa associativa executada pelo estágio **DS2** para selecionar o possível trace ou instrução redundante é ocultada pelas operações efetuadas pelo *Estágio de Decodificação* para despacho de instruções (i.e., verificação e geração de tag). É importante observar, que as pesquisas associativas executadas pelos estágios **DS1** e **DS2** do mecanismo *DTM*, são completamente sobrepostas, pois operam sobre diferentes campos da tabela de memorização [71]. As tabelas de memorização podem ser implementadas de modo a prover acessos que ocorram com o dobro da velocidade do processador [37, 25], suportando deste modo maiores latências (este artifício tem sido usado para aumentar por divisão de tempo o número de portas de acesso aos caches [55]). Finalmente, as operações sobre mapas de bits de contexto de entrada/saída e buffers temporários, executadas pelo estágio DS3, podem ser ocultadas por operações do *Estágio de Entrega*.

A principal demanda de hardware que poderia ser ocasionada pelo mecanismo *DTM*, recai sobre o arquivo de registradores, pois portas de leitura adicionais seriam necessárias para permitir acessos simultâneos efetuados pelo estágio **DS2** do *DTM* e pelo *Estágio de Decodificação*. Entretanto, modernas tecnologias de fabricação tornam viável a implementação de um arquivo de registradores multiportas sem limitação do ciclo de clock ou consumo de muita área de silício. Em [69] é avaliado o efeito (atraso no acesso e área ocupada em silício) da quantidade de portas implementadas em um arquivo de registradores de um processador *VLIW* para processamento de sinais de vídeo. Neste trabalho foi mostrado que o atraso no acesso ao arquivo de registradores é ligeiramente dependente do número de portas, embora a área possa aumentar por um fator de dez quando o número de portas varia na razão de três para doze. Usando tecnologia 0.25μm e 4-camadas de metal, foi concluído que um arquivo de registradores com 12 portas e com organização de 128×16-bit, ocupa 3.0mm^2 e opera a 650MHz. Arquivo de registradores grandes e com muitas portas, são apresentados em projetos atuais e em projetos relativamente antigos

[41, 68]. Por exemplo, o *SPARC64* [68] é fabricado com especificação de projeto de 0.4μm e 4 camadas de metal, e possui um arquivo de registradores com 14 portas de acesso e organização de 116×64-bit. Adicionalmente, como será mostrado no Capítulo 5, para os programas do *SPEC95*, nenhuma porta adicional foi requerida ao arquivo de registradores quando da avaliação do mecanismo *DTM*.

4.4 Diferenças entre o *DTM* e os esquemas de reuso S_{n+d} e *BHB*

Esta seção pretende estabelecer as diferenças básicas entre o mecanismo *DTM* e os mecanismos S_{n+d} [63] e *BHB* [35]. O mecanismo *DTM* aqui apresentado, se destaca dos demais mecanismos pelas seguintes considerações:

- Não explora a redundância de instruções com efeitos colaterais, principalmente instruções de acesso à memória, evitando deste modo, mecanismos complexos para validação destas;

- Explora computações redundantes com granularidade em nível de traces, ou seja, não se limita às fronteiras de blocos básicos;

- Fornece uma estrutura comum, na qual diferentes heurísticas podem ser adotadas para identificar construtivamente traces;

- Apresenta adaptabilidade com relação à redundância, ou seja, enquanto os outros mecanismos procuram explorar redundância considerando limites intuitivos como cadeias de dependências ou blocos básicos, o *DTM* considera apenas o comportamento redundante como fator de seleção e construção dos traces. Esta consideração é decorrente da estrutura implementada no *DTM*, pois este considera duas tabelas com níveis de volatilidade completamente diferentes. Enquanto a *Memo_Table_G* é extremamente volátil pois captura quaisquer instruções que pertencem ao domínio de instruções válidas, a *Memo_Table_T* em contrapartida, é pouco volátil, dado que o processo de construção de traces depende da redundância de instruções simples.

As seguintes exposições, revelam as diferenças entre o *DTM* e os esquemas de reuso S_{n+d} e *BHB*:

• Para determinar se uma cadeia de instruções pode ser reusada ou não, o esquema S_{n+d} deve verificar para cada instrução da cadeia, se a relação de dependência existente é mantida inalterada desde a última execução da instrução [63]. De outro modo, o mecanismo *DTM* não necessita verificar encadeamentos que interligam uma cadeia de instruções dependentes, visto que esta informação é implícita ao trace;

• No esquema S_{n+d}, as relações de dependência são avaliadas apenas para as instruções que estão presentes no buffer de instruções [63]. Portanto, a largura de despacho[3] restringe o tamanho das cadeias a serem reusadas (isto é, o número de instruções que podem ser reusadas no mesmo ciclo). Contrariamente, o mecanismo *DTM* é desacoplado deste parâmetro definido pela microarquitetura base. O número de instruções reusadas por um trace redundante é independente da largura de despacho. Esta característica torna o *DTM* atrativo, mesmo para arquiteturas com largura de despacho relativamente estreitas [38];

• Além de ser restrito a largura de despacho, o esquema S_{n+d} é limitado pelo alinhamento das instruções no buffer de instruções. Mesmo para o caso em que as cadeias de instruções possuam um número de instruções menor que a largura de despacho, estas podem não ser completamente reusadas, dependendo da posição da primeira instrução da cadeia no buffer de instruções. Mais precisamente, para uma cadeia de instruções de comprimento l e sendo w a largura de despacho, para $l \leq w$, a cadeia de instruções dependentes é completamente reusada em um único ciclo, apenas se a sua primeira instrução ocupa uma posição p no buffer de instruções e de modo que $p \leq w - l$. Contrariamente, o *DTM* não sofre problemas de alinhamento de instruções no buffer de instruções;

• De modo a permitir a verificação de dependências, o esquema S_{n+d} deve guardar todas as instruções compreendidas pela cadeia reusável no *Reuse Buffer* [63]. Como resultado, o número de cadeias reusadas que podem ser simultaneamente cacheadas, é sensível ao número de entradas do *Reuse Buffer*. De outro modo, o *DTM* não conserva as instruções abrangidas pelos traces montados, e as informações que representam o trace por completo, ocupam uma simples entrada de *Memo_Table_T*. Portanto, o *DTM* potencialmente armazena mais instruções reusáveis do que o es-

[3]Largura de despacho corresponde ao número máximo de entrada no buffer de instruções.

quema S_{n+d} (dado uma *Memo_Table_T* e um *Reuse Buffer* com o mesmo número de entradas);

• No esquema S_{n+d}, instruções dinâmicas são inseridas em uma mesma estrutura que conserva a informação de reuso (*Reuse Buffer*), isto a torna sensível a política adotada para gerenciar a substituição de instruções. Políticas que possuem implementação simples (políticas Round-Robin e FIFO), podem substituir instruções que estão encadeadas. Portanto, a política de substituição de instruções pode encurtar, fragmentar ou mesmo excluir completamente uma cadeia de instruções reusáveis. De modo contrário, no mecanismo *DTM*, substituições de instruções não interferem no reuso de traces, dado que duas estruturas são empregadas para colecionar instruções dinâmicas (*Memo_Table_G*) e guardar a informação de reuso dos traces (*Memo_Table_T*). Adicionalmente, esta separação permite a adoção de diferentes políticas para substituição de instruções e de substituição de traces;

• O esquema S_{n+d} não é capaz de tratar múltiplos desvios de controle de execução, não permitindo explorar reuso entre blocos básicos, pois o reuso especulativo (*Squash Reuse*) não permite o reuso de instruções de desvio condicional;

• O *DTM* pode explorar a antecipação dos resultados de instruções que ainda não foram buscadas. Esta consideração pode ser exemplificada por um trace possuindo três instruções, e considerando que a sua primeira instrução ocupa a última posição do buffer de instruções. Para este caso, o *DTM* é capaz de antecipar o reuso de instruções que ainda estão sendo buscadas (as duas instruções subseqüentes à primeira);

• O mecanismo *BHB* explora redundância com granularidade em nível de blocos básicos e limita-se a estas fronteiras;

• O mecanismo *BHB* necessita da intervenção do compilador para detectar e marcar elementos do contexto de saída que não serão referenciados em computações posteriores. Este procedimento possui como objetivo, cobrir deficiências não tratadas pelo mecanismo e que podem adicionar uma quantidade maior de hardware. Entretanto, as marcações de código inserem o inconveniente da portabilidade de código (*legacy code*);

• O mecanismo *BHB* exige no mínimo, o acréscimo de 4 portas de leitura no cache

de dados *L1*, de modo a executar antecipadamente as instruções de leitura à memória (quando estas estão presentes no bloco básico), garantindo assim a consistência da mesma.

O exemplo a seguir, apresenta a aplicação dos mecanismos S_{n+d} e *DTM* à mesma seqüência de código da Figura 4.16(a). Esta apresenta o fluxo de execução de uma seqüência de código que será executada pelos processadores configurados. A Figura 4.16(b) esboça a disposição das instruções da seqüência de código no cache de instruções. A Figura 4.16(c) apresenta a execução da mesma seqüência de código, considerando um processador superescalar (substrato), o mesmo processador superescalar suportando o mecanismo S_{n+d} e o mesmo processador superescalar suportando o mecanismo *DTM*. Para a configuração substrato é suposto que a predição do desvio foi correta e que nenhuma penalidade será inserida ao se redirecionar o fluxo de execução para o endereço alvo da predição. Para as outras configurações é suposto que os registradores *r2* e *r3* que determinam a redundância da seqüência de código, estão instanciados com valores já observados anteriormente, e que a seqüência (ou idealmente, fragmentos redundantes da seqüência para o caso do S_{n+d}) encontra-se armazenada em ambas tabelas de memorização (*Reuse Buffer* e *Memo_Table_T*). Será considerado também para este exemplo, que para qualquer instrução **j** que dependente de uma instrução **i** executada no ciclo t, poderá ser enviada para execução no ciclo $t+1$. É importante notar, que esta última consideração é indiferente para o mecanismo *DTM*, porém sem a mesma, o processador substrato e o mesmo processador incorporando o S_{n+d}, poderiam sofrer uma dramática redução de *ipc*.

Analisando-se as três execuções expostas pela Figura 4.16(c), observa-se que: foram necessários 8, 6 e 4 ciclos para executar completamente a mesma seqüência de código nos processadores substrato, S_{n+d} e *DTM* respectivamente, ou seja, os mecanismo S_{n+d} e *DTM* efetuam a mesma tarefa consumindo 3/4 e 1/2 respectivamente, do número de ciclos dispendidos pelo processador substrato; o *DTM* apresenta uma reduzida ocupação de recursos do processador, pois: no estágio de busca buscou **3 (três)** instruções (não necessitou buscar a instrução L); decodificou apenas **2 (duas)** instruções (I e J no estágio de decodificação); enviou apenas o contexto de saída que escreve em **2 (dois)** registradores *r1* e *r3* (ocupando deste modo somente 2

entradas no buffer de emissão que atualizam o contexto de saída *wr1* e *wr3*); não executou nenhuma instrução; e no estágio de entrega atualizou somente as **2** (duas) instruções do contexto de saída escalonadas na emissão. Em contrapartida, os respectivos valores para processador base foram 4,4,4,4 e 4, e para o processador suportando o mecanismo S_{n+d} foram 4,4,4,0 e 4. É importante observar que o ganho de desempenho de um processador superescalar incorporando um mecanismo de reuso não se efetivará apenas considerando a quantidade de instruções reusadas, mas também pelo efeito da redução de recursos requisitados em decorrência do reuso.

CÓDIGO

```
I→ 100   add r1,r2,#5
J→ 104   sll r3,r3,r1
K→ 108   ble r3, r0, #1FC
. . .
L→ 308   add  r1,r3,#2048
```

(a)

Disposição das instruções no cache de instruções

(b)

Pipeline		Busca	Decodificação e Despacho	Emissão	Execução	Entrega
Base Superescalar	1	I,J				
	2	K	I,J			
	3	L	K	I,J		
	4		L	J,K	I	
	5			K,L	J	I
	6			L	K	J
	7				L	K
	8					L
S_{n+d}	1	I,J				
	2	K	I,J			
	3	L	K	I,J		
	4		L	K		I,J
	5			L		K
	6					L
DTM	1	I,J				
	2	K	I,J			
	3			wr1,wr3		
	4					wr1,wr3

(c)

Figura 4.16: Execução da mesma seqüência de código pelos processadores: substrato, S_{n+d} e *DTM*

Capítulo 5

Base Experimental, Resultados e Avaliações

5.1 Base Experimental

5.1.1 Ambiente de Simulação

Para efetuar os experimentos de avaliação do mecanismo proposto, foi utilizado o simulador *sim-outorder* do *simplescalar 2.0 tool suite* [16]. O simulador baseia-se na arquitetura descrita em [62], e sofreu algumas modificações para representar a arquitetura substrato proposta no capítulo 4. Ao simulador foram incorporados os estágios e tabelas de memorização do mecanismo *DTM*. Os recursos computacionais utilizados para simulação foram compostos por estações *SPARC Sun Ultra 1, 5 e 10* sob o controle do sistema operacional *SunOS versão 5.6*.

5.1.2 Programas de Teste

Todos os programas de teste do *SPECInt95* e oito programas do *SPECFp95* foram utilizados para a realização dos experimentos. A tabela 5.1 e 5.2, apresentam a relação dos programas com as respectivas entradas utilizadas e o número de instruções simuladas (estas incluem a fase de inicialização). Alguns programas não foram completamente finalizados, entretanto, considerando as entradas utilizadas, foi executado um número significativo de instruções, habilitando-os para a análise posterior. Convém mencionar que para alguns dos programas do *SPECFp95*, al-

guns dos valores de entrada que ditam o número de iterações, foram modificados para que fosse obtido o máximo possível de execução. Estas alterações são explicitadas conjuntamente com as entradas na tabela 5.2. Todos os programas de teste foram compilados utilizando o *gcc-2.6.3* e *glibc-1.09* do *simplescalar 2.0 tool suite* e com a opção de otimização -*O3* ativada .

Tabela 5.1: Programas utilizados nos experimentos, *SPECInt95*.

Benchmark	**Entradas**		**número de inst. simuladas**	
cc1	expr.i	(ref)	finalizado	236.438.809
compress	test.in	(train)	finalizado	35.684.602
go	9stone21	(ref)	finalizado	132.917.038
ijpeg	vigo.ppm	(train)	finalizado	249.725.697
li	deriv.lsp	(ref)	finalizado	537.745.837
m88ksim	ctl.raw	(test)	finalizado	492.996.053
perl	primes.in	(ref)	parcial	300.000.000
vortex	vortex.in	(ref)	parcial	300.000.000

Tabela 5.2: Programas utilizados nos experimentos, *SPECFp95*.

Benchmark	**Entradas**		**número de inst. simuladas**	
applu	applu.in	(train)	finalizado	531.902.884
apsi	apsi.in	(train)	parcial	100.000.000
hydro2d	hidro2d.in, ISTEP=5	(train)	parcial	500.000.000
su2cor	su2cor.in, LSIZE= 4 4 4 4	(train)	finalizado	326.481.936
swim	swim.in, X=10 e Y=10	(train)	finalizado	199.790.520
tomcatv	tomcatv.in, ITER=50	(train)	finalizado	136.520.817
turb3d	turb3d.in, NSTEP=4	(train)	parcial	500.000.000
wave5	wave5.in	(train)	parcial	500.000.000

Para fornecer informações adicionais sobre os programas avaliados, as tabelas 5.3 e 5.4 apresentam uma distribuição percentual por tipo de instruções executadas para os programas do *SPEC95*. As tabelas apresentam percentuais para instruções de acesso à memória (*load/store*), desvios (condicionais, incondicionais, diretos, relativos, chamada e retorno de subrotinas), aritméticas para valores inteiros, aritméticas para valores de ponto flutuante, e outras (chamadas ao sistema operacional, nops, etc...).

Tabela 5.3: Distribuição percentual das instruções executadas (por tipo).

SPECInt95					
Benchmark	**Load/Store**	**Desvios**	**Inst. Inteiros**	**Inst. Pf.**	**Outras**
cc1	36.97%	20.81%	42.21%	0.00%	0.01%
compress	37.42%	17.22%	44.83%	0.52%	0.01%
go	27.62%	15.22%	57.16%	0.00%	0.00%
ijpeg	27.24%	12.60%	59.66%	0.00%	0.00%
li	46.90%	21.83%	31.27%	0.00%	0.00%
m88ksim	25.81%	23.00%	51.19%	0.00%	0.00%
perl	47.62%	19.17%	31.65%	1.56%	0.00%
vortex	53.46%	15.87%	30.67%	0.00%	0.00%

Tabela 5.4: Distribuição percentual das instruções executadas (por tipo).

SPECFp95					
Benchmark	**Load/Store**	**Desvios**	**Inst. Inteiros**	**Inst. Pf.**	**Outras**
applu	25.51%	3.35%	48.80%	22.33%	0.01%
apsi	31.48%	5.48%	43.63%	19.41%	0.00%
hydro2d	25.12%	12.58%	40.60%	21.70%	0.00%
su2cor	30.07%	7.34%	47.62%	14.97%	0.00%
swim	29.47%	6.37%	37.91%	26.25%	0.00%
tomcatv	26.65%	6.57%	54.82%	11.96%	0.00%
turb3d	22.47%	5.28%	57.58%	14.26%	0.41%
wave5	30.64%	8.24%	42.26%	18.84%	0.02%

5.1.3 Parâmetros Arquiteturais do Processador Simulado

A tabela 5.5 explicita a configuração dos parâmetros arquiteturais adotados para o processador simulado.

5.1.4 Parâmetros Arquiteturais do Mecanismo *DTM*

A tabela 5.6 apresenta os parâmetros e seus respectivos valores que configuram o mecanismo *DTM* para a seqüência de resultados que serão apresentados.

Tabela 5.5: Configuração do processador superescalar substrato.

Busca de Instruções	4 instruções por ciclo. Apenas um desvio tomado por ciclo. Pode ultrapassar as fronteiras de uma linha de cache no mesmo ciclo.
Cache de Instruções	16Kbytes, associativo, 2 por conjunto, 32 bytes por linha, latência de 6 ciclos para acessos sem sucesso ao cache L1 e latência de 20 ciclos para acessos sem sucesso ao cache L2.
Preditor de desvios	Bimodal, 2048 entradas, pode predizer vários desvios por ciclo.
Mecanismo de Execução	Execução fora-de-ordem, buffer de emissão/reordenação com 16 entradas e mais uma fila de load/store com 8 entradas. Loads são executados após serem conhecidos todos os endereços de stores precedentes. Loads são servidos por stores acessando o mesmo endereço se ambos estiverem na fila de load/store. Suporte para execução especulativa.
Registradores Arquiteturais	32 registradores inteiros, 32 registradores de ponto flutuante, registradores hi , lo e fcc.
Unidades Funcionais	4 ULAs de inteiros, 2 unidades de load/store, 4 somadores de pf, 1 MULT/DIV inteiro, 1 MULT/DIV pf.
Latência das Unid. Funcionais	ULA-inteiros/1, load/store/1, int mult/3, int div/20, somadores pf./2, mult pf./4, div pf./12, sqrt pf./24.
Cache de Dados	16Kbytes, associativo, 4 por conjunto, 32 bytes por linha, latência de 6 ciclos para acessos sem sucesso ao cache L1 e latência de 20 ciclos para acessos sem sucesso ao cache L2. Não bloqueante.

5.2 Resultados

5.2.1 Métrica

As seguintes equações determinaram os valores obtidos nas simulações:

itot - Número total de instruções executadas.

ipc - Número médio de instruções executadas (dentre as entregues) por ciclo de clock.

ir - Número de instruções reusadas pelo mecanismo de reuso.

speedup - Aceleração de desempenho.

HM - Média harmônica.

Tabela 5.6: Parâmetros arquiteturais do mecanismo *DTM*.

Parâmetros do *DTM*	**Valores**
Contexto de entrada	até 6 registradores
Contexto de saída	até 6 registradores
Numero de desvios tratados	até 10 desvios
Tamanho. máx dos Traces	até 16 instruções
Heurística	Repetição de instruções simples
Conj. de Seleção	Instruções: Aritméticas, Lógicas, Cálculo de endereços simples, Desvios Incondicionais e Condicionais Movimentação de dados, Chamada e Retorno de Subrotinas
Política de atualização tabelas de memorização	LRU

AM - Média aritmética.

O Percentual de Reuso é dado por :

$$ir/itot$$

O Percentual de Aceleração de Desempenho é dada por:

$$speedup = ipc_{DTM}/ipc_{base}$$

Onde ipc_{DTM} e ipc_{base} são considerados respectivamente para um processador incluindo e não incluindo o mecanismo *DTM*.

As Médias são dadas pelas seguintes expressões:

$HM = n(\sum_{i=1}^{n}(1/S_i))^{-1}$

$AM = (\sum_{i=1}^{n} S_i)/n$, S_i indica o elemento a ser considerado pela média.

5.2.2 Reuso e Aceleração

Para esta fase inicial de avaliações, serão apenas considerados os programas do *SPECInt95*. Este posicionamento decorre da própria concepção do mecanismo *DTM*, pois o mecanismo foi inicialmente proposto e definido exclusivamente para obtenção

ganhos de performance sobre programas que manipulam valores inteiros. A definição do mecanismo não inclui o tratamento de instruções de ponto flutuante e conseqüentemente não faz referências ao hardware necessário pra suportá-las (que incorreria em uma grande elevação dos custos e da complexidade envolvida). Um outro fator que foi decisivo para a exclusão de instruções de ponto flutuante, considera a baixa localidade de valores apresentada por estas nos trabalhos [44, 29], aliada a existência de técnicas dedicadas para este fim, e com simplicidade de implementação [50, 18]. Apesar do exposto, será visto adiante, que os programas do *SPECFp95* beneficiam-se do mecanismo *DTM* mesmo sem a estensão deste.

A primeira medida a ser exposta, considera o número de entradas aplicadas às tabelas *Memo_Table_G* e *Memo_Table_T*, este parâmetro de configuração determina os pontos de saturação do mecanismo. Os números da legenda do gráfico das Figuras 5.1 e 5.2, indicam o número de entradas atribuídas a ambas tabelas *Memo_Table_G* e *Memo_Table_T*, ou seja, o número de entradas de *Memo_Table_G* é igual ao número de entradas de *Memo_Table_T* para cada valor apresentado. Ex: 256, indica que as simulações consideraram *Memo_Table_G* com 256 entrada e *Memo_Table_T* com 256 entradas.

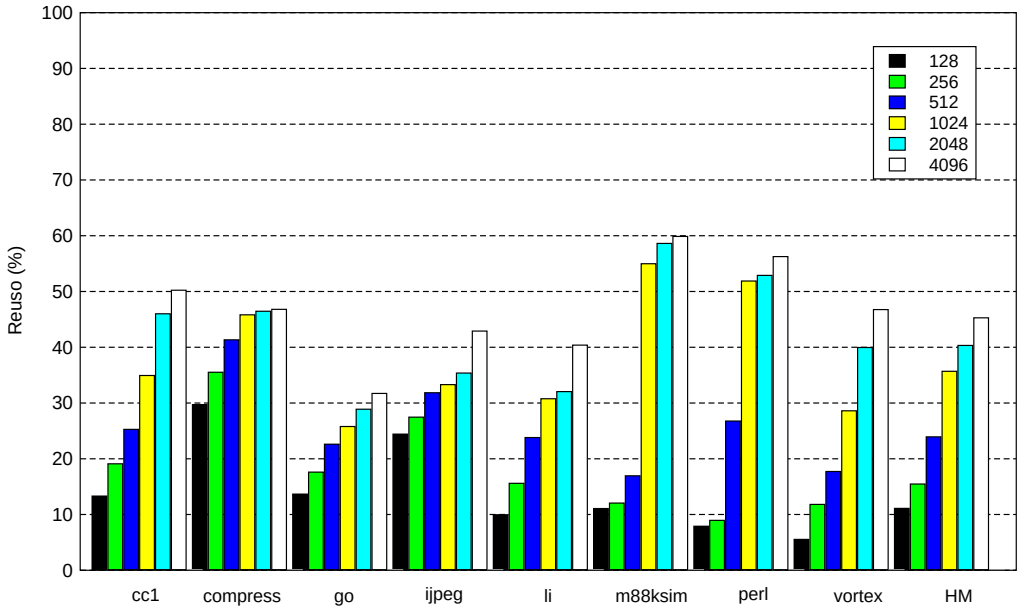


Figura 5.1: Variações no reuso observado considerando tabelas de memorização com um mesmo número de entradas.

Os resultados plotados na Figura 5.1, apresentam o percentual de reuso obtido

quando varia-se o número de entradas nas tabelas *Memo_Table_G* e *Memo_Table_T*. Os valores variam de 7.9% a 29.7% para *vortex* e *compress* respectivamente, para tabelas com 128 entradas. Para tabelas com 4096 entradas, os valores variam de 31.7% a 59.8% para *go* e *m88ksim* respectivamente. A média harmônica obtida, varia de 10.5% a 45% para tabelas com 128 e 4096 entradas respectivamente.

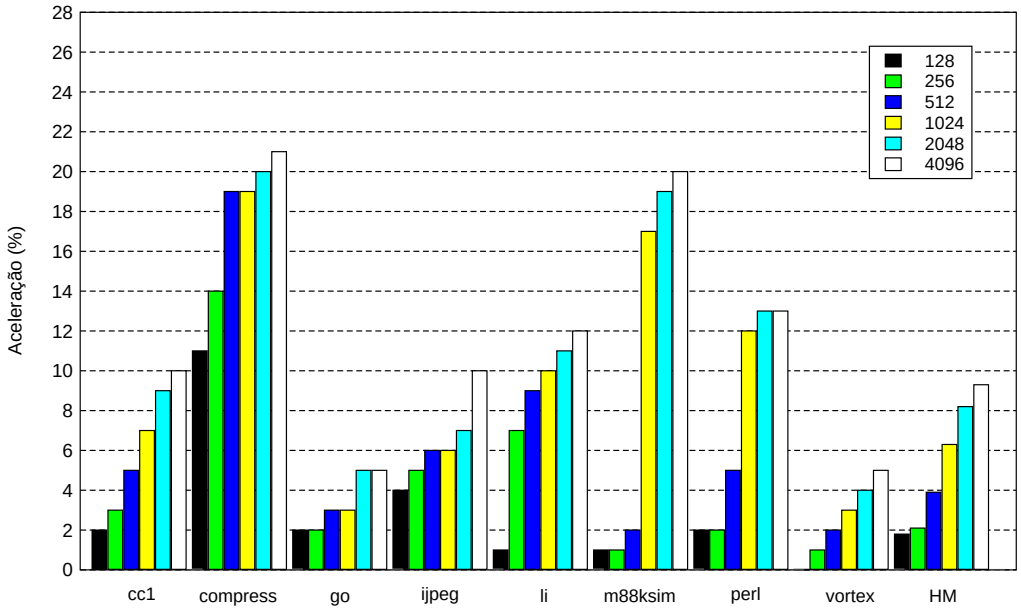


Figura 5.2: Variações de aceleração de desempenho observada considerando tabelas de memorização com um mesmo número de entradas.

O gráfico da Figura 5.2, apresenta a variação de aceleração de desempenho de um processador incorporando o mecanismo *DTM*, considerando como base o mesmo processador não incluindo o mecanismo de reuso. Analisando os resultados apresentados, observa-se o percentual de aceleração de desempenho obtido quando varia-se o número de entradas nas tabelas *Memo_Table_G* e *Memo_Table_T*. Os ganhos de performance em média harmônica variaram em 1.8%, 2.1%, 3.4%, 6.3%, 8.2% e 9.4% para as tabelas de memorização (*Memo_Table_G* e *Memo_Table_T*) com 128, 256, 512, 1024, 2048 e 4096 entradas respectivamente. Especificamente, os valores variam de 0% a 11% para *vortex* e *compress* respectivamente, considerando tabelas com 128 entradas; e variam de 5% a 21% para *vortex* e *compress* respectivamente, considerando tabelas com 4096 entradas. Observa-se ainda, que para o programa *m88ksim*, o número de entradas afeta a aceleração acentuadamente, principalmente

quando esta varia de 512 (2%) para 1024 entradas (17%).

5.2.3 Custo-Efetividade

Os resultados anteriores são baseados em configurações para as quais *Memo_Table_G* e *Memo_Table_T* possuem o mesmo número de entradas. O conjunto de resultados a serem expostos a seguir, apresentarão o efeito do tamanho de cada tabela de memorização na performance do processador incorporando o mecanismo *DTM*. Para as medidas a seguir, uma das tabelas será fixada em 4k entradas e a outra irá variar no sentido crescente (128, 256, 512, 1024, 2048, 4096), se aproximando do número de entradas da tabela fixa. Este procedimento será aplicado para as duas tabelas de memorização.

A Figura 5.3 apresenta os resultados das simulações, considerando um número fixo de 4096 entradas para *Memo_Table_T* e variando-se o número de entradas de *Memo_Table_G* entre 128 e 4096. Neste experimento, os ganhos de performance em média harmônica variaram em 0.6%, 2.8%, 4.4%, 7.1%, 8% e 9.4% para *Memo_Table_G* com 128, 256, 512, 1024, 2048 e 4096 entradas respectivamente. Analisando os resultados plotados, observa-se que os ganhos de performance começam a se apresentar mais apreciáveis quando *Memo_Table_G* atinge 1024 entradas. Neste ponto, os ganhos de performance variam de 3.3% para o *vortex* a 19% para o *compress* e *m88ksim*, e possui o valor de média harmônica igual a 7.1%. Os programas *m88ksim* e *perl* se beneficiaram desta configuração, pois observaram transições de performance de 2.5% e 5.1% para 19% e 12% respectivamente. Este gráfico apresenta valores bastante próximos dos valores obtidos no gráfico da Figura 5.2. Esta característica reflete a importância de *Memo_Table_G* no mecanismo *DTM*. Este comportamento era esperado, desde que *Memo_Table_G* é a fornecedora de instruções redundantes que irão compor os traces, e o número de traces capturados e reusados dependem diretamente da taxa de acertos em *Memo_Table_G*.

A Figura 5.4 apresenta os resultados das simulações considerando um número fixo de 4096 entradas para *Memo_Table_G* e variando-se o número de entradas de *Memo_Table_T* entre 128 e 4096. Neste experimento, os ganhos de performance em média harmônica variaram em 5.4%, 7.2%, 7.9%, 8.2%, 8.8% e 9.4%, para *Memo_Table_T* com 128, 256, 512, 1024, 2048 e 4096 entradas respectivamente. Ana-

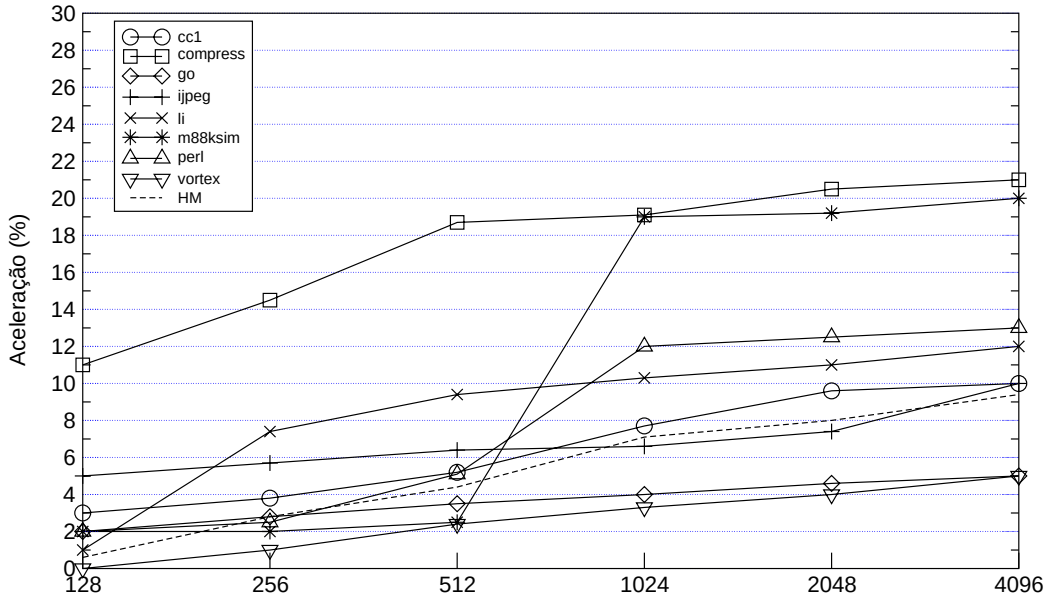


Figura 5.3: Variações de aceleração de desempenho, considerando *Memo_Table_T* fixa em 4k entradas e o efeito da variação de *Memo_Table_G*

lisando os resultados, observa-se que os ganhos de performance são significativos quando se alterna a configuração de *Memo_Table_T* com 128 entradas para 256 entradas (média harmônica de 5.4% e 7.2% respectivamente). Os ganhos de performance entre as configurações de *Memo_Table_T* com 256 e 1024 entradas (inclusive), apresentam um crescimento razoável (média harmônica de 7.2% a 8.2%). A partir de configurações superiores a 1024 entradas, os ganhos de performance não são tão acentuados a ponto de compensarem o custo de hardware (média harmônica de 8.2% a 9.4% para 2048 e 4096 entradas respectivamente), pois adicionam ganhos de performance de no máximo 1.2%.

Para avaliar melhor o comportamento do mecanismo com relação ao número de entradas de *Memo_Table_T*, foram simuladas configurações onde o número de entradas de *Memo_Table_G* foi fixado em 1024 (análise da Figura 5.3) e o número de entradas em *Memo_Table_T* variou de 128 a 4096. A Figura 5.5 apresenta os resultados de simulações. Verifica-se neste experimento, que os ganhos de performance em média harmônica variaram em 4.7%, 5.6%, 5.8%, 6.3%, 6.5% e 7.1% para *Memo_Table_T* com 128, 256, 512, 1024, 2048 e 4096 entradas respectivamente. Considerando os resultados obtidos e o interesse de minimizar o custo, pode-se escolher um valor entre 256 e 1024 entradas para compor a *Memo_Table_T*.

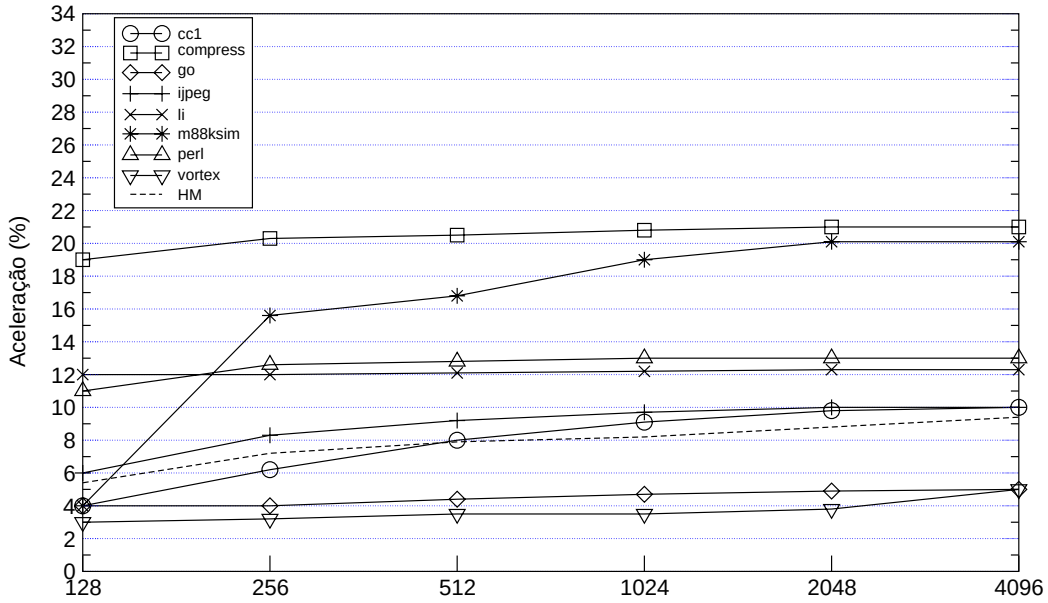


Figura 5.4: Variações de aceleração de desempenho, considerando *Memo_Table_G* fixa em 4k entradas e o efeito da variação de *Memo_Table_T*.

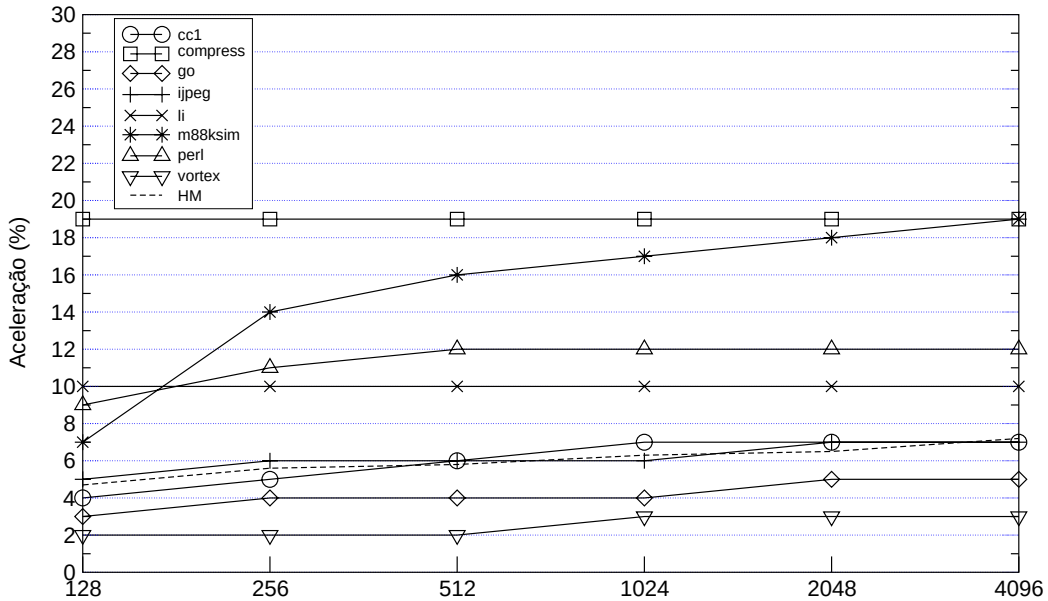


Figura 5.5: Variações de aceleração de desempenho, considerando *Memo_Table_G* fixa em 1024 entradas e o efeito de variação de *Memo_Table_T*.

Os valores até aqui plotados confirmam a importância do número crescente de entradas em *Memo_Table_G* e induzem a escolha de *Memo_Table_T* e *Memo_Table_G* ambas com 1024 entradas, onde se explora um ganho de performance razoável (média harmônica de 6.3%) sem grandes investimentos em custo.

A tabela 5.7 e o gráfico da Figura 5.6, resumem os experimentos efetuados anteriormente (considerando as médias harmônicas obtidas).

Tabela 5.7: Variando o número de entradas nas tabelas de memorização.

Memo_Table_T	*Memo_Table_G*	**128**	**256**	**512**	**1024**	**2048**	**4096**
varia	varia	1.8	2.1	3.4	6.3	8.2	9.4
fixa em 4k	varia	0.6	2.8	4.4	7.1	8	9.4
varia	fixa em 4k	5.4	7.2	7.9	8.2	8.8	9.4
fixa em 1k	varia	4.7	5.6	5.8	6.3	6.5	7.1

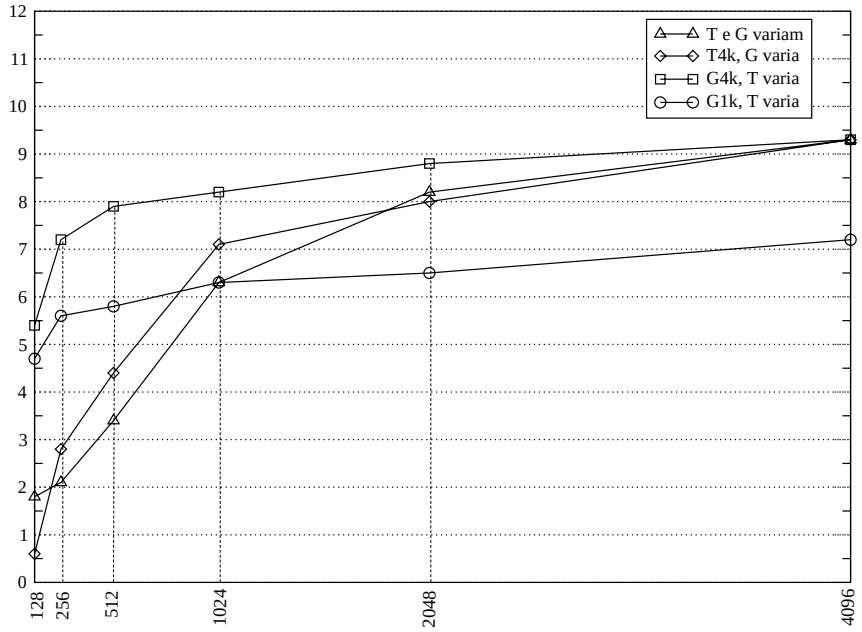


Figura 5.6: Variações de aceleração de desempenho obtidas pelas configurações simuladas

Considerando a análise efetuada na Figura 5.3 e a aparente menor importância do efeito do número de entradas de *Memo_Table_T* como mostrado nas Figuras 5.4 e 5.5. Pode-se induzir que a performance do *DTM* é decorrente primariamente do reuso de instruções individuais observadas em *Memo_Table_G*. Para avaliar

realmente o efeito do reuso de traces, duas configurações para o *DTM* foram propostas e simuladas. Inicialmente foi assumido uma capacidade de armazenamento total de 784 Kbits para as tabelas do *DTM* (esta quantidade de bits foi escolhida para prover futuras comparações com outros mecanismos). Na primeira configuração, o *DTM* foi configurado com 512 entradas para *Memo_Table_T* e o restante dos bits (do total de 784Kbits) foram alocadas para *Memo_Table_G*, produzindo para esta última, uma configuração com 4672 entradas. Uma segunda configuração foi construída removendo-se a *Memo_Table_T* e atribuindo o total dos 784 Kbits para *Memo_Table_G*, produzindo para esta uma configuração com 6143 entradas. Esta configuração será denominada *Reuso Simples* (esta segunda configuração só irá reusar instruções simples identificadas em *Memo_Table_G*). A escolha de *Memo_Table_T* com 512 entradas foi decorrente das medidas expostas na tabela 5.7, considerando esta, observa-se que a diferença entre *Memo_Table_T* com 512 e 1024 entradas (quando *Memo_Table_G* é fixada em 4k entradas) é praticamente insignificante (0.7% de diferença na média harmônica), logo a escolha de uma *Memo_Table_T* com um menor número de entradas apresenta-se mais atraente em função do custo de implementação, além de prover uma *Memo_Table_G* com várias entradas e contribuindo deste modo para a identificação e construção de um maior número de traces. É importante notar que a escolha de uma *Memo_Table_T* com 1024 entradas, forçaria *Memo_Table_G* a possuir somente 3233 entradas, o que produziria um valor de ganho de performance menor que 8.2%, visto que este valor corresponde a uma *Memo_Table_G* com 4k entradas.

As Figuras 5.7 e 5.8, apresentam os resultados de uma comparação efetuada entre o *Reuso simples* e o *DTM*. Nesta comparação são avaliados os percentuais de reuso obtidos pela aplicação dos mecanismos configurados. Os resultados plotados demonstram o efeito do reuso de traces sobre o reuso de instruções individuais. Todos os programas avaliados apresentaram um maior percentual de reuso para a configuração *DTM*. Os programas do *SPECInt95* apresentaram uma média harmônica de 33% e 44% para o *Reuso simples* e *DTM* respectivamente, enquanto os programas do *SPECFp95* apresentaram uma média harmônica de 25% e 30% para o *Reuso simples* e *DTM* respectivamente.

As Figuras 5.9 e 5.10, apresentam os resultados de uma comparação efetuada

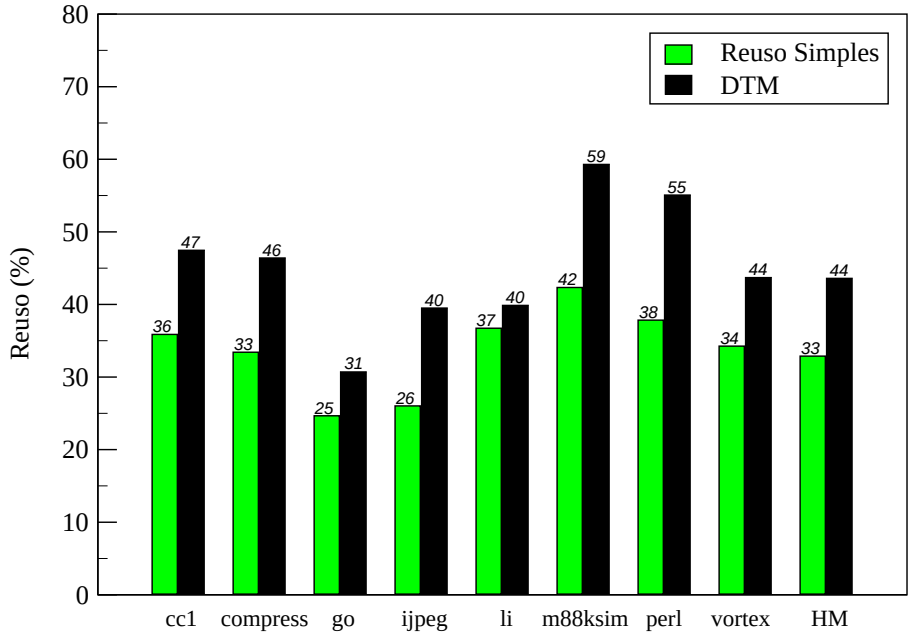


Figura 5.7: Comparação do percentual de reuso entre *Reuso Simples* e *DTM*, *SPECInt95*.

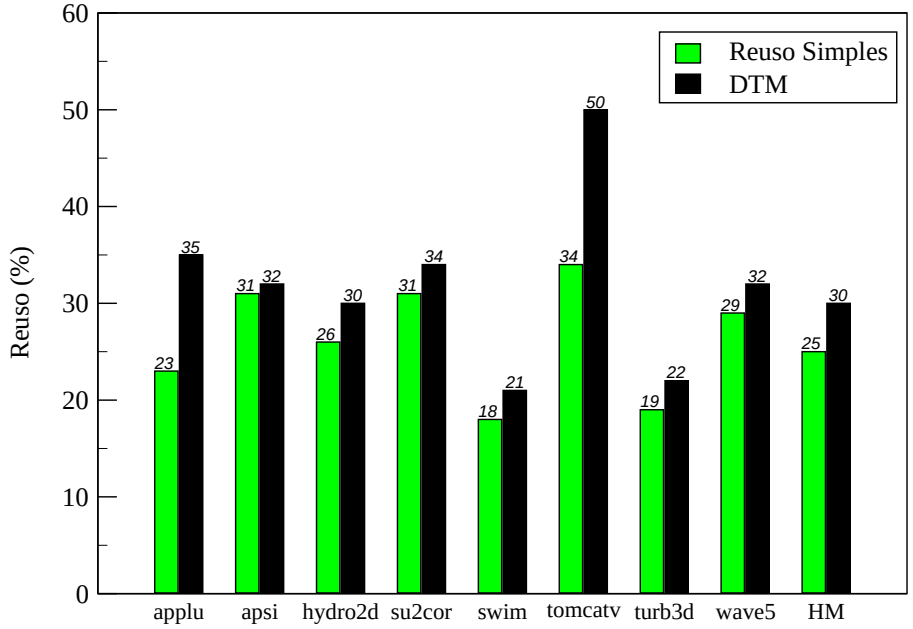


Figura 5.8: Comparação do percentual de reuso entre *Reuso Simples* e *DTM*, *SPECFp95*.

entre o *Reuso Simples* e o *DTM*, onde são avaliados os ganhos de performance obtidos pela aplicação dos mecanismos configurados. Os resultados plotados nos gráficos demonstram o efeito do ganho de performance de traces sobre instruções individuais. Todos os programas avaliados apresentaram aumentos expressivos nos ganhos de performance. Os programas do *SPECInt95* apresentaram uma média harmônica de 3.9% e 8.4% para o *Reuso simples* e *DTM* respectivamente, enquanto os programas do *SPECFp95* apresentaram uma média harmônica de 3.5% e 7% para o *Reuso simples* e *DTM* respectivamente. As medidas apresentadas conferem ao *DTM* um desempenho 100% superior ao *Reuso simples* e demonstram a importância do reuso dos traces armazenados em *Memo_Table_T*.

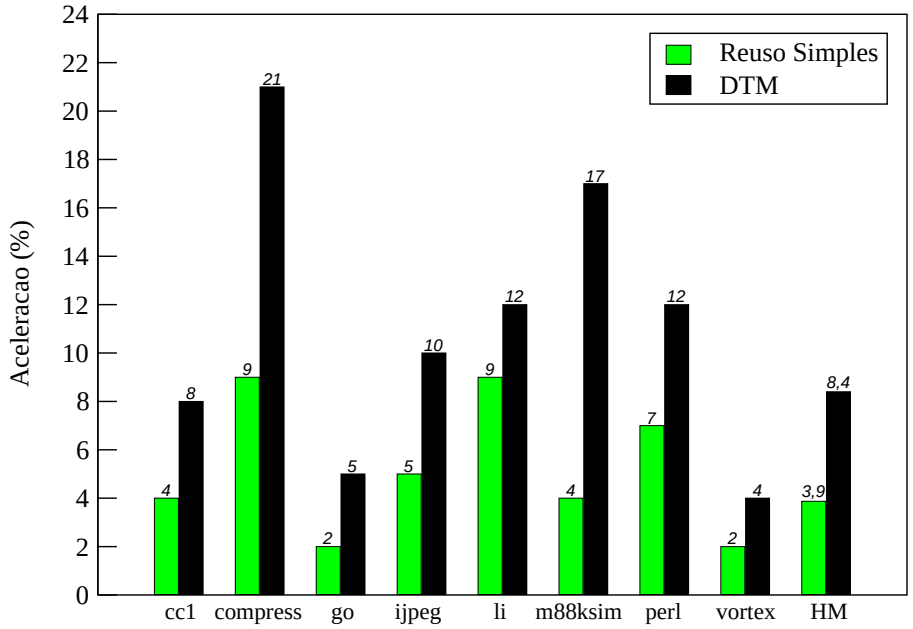


Figura 5.9: Comparação do percentual de ganhos de performance entre *Reuso Simples* e *DTM*, *SPECInt95*.

No cômputo geral, o *DTM* apresentou resultados superiores ao reuso de instruções simples. Entretanto, as medidas apresentadas descrevem um cenário curioso e não intuitivo. Observa-se diretamente pelas medidas, que a diferença entre os percentuais de reuso obtidos pela comparação *Reuso Simples* e *DTM* não são proporcionais aos ganhos de performance obtidos. Este comportamento é decorrente de vários fatores, entre eles: o tipo de instrução reusada e sua contribuição no ganho de performance; disponibilidade de recursos quando da efetuação de reuso;

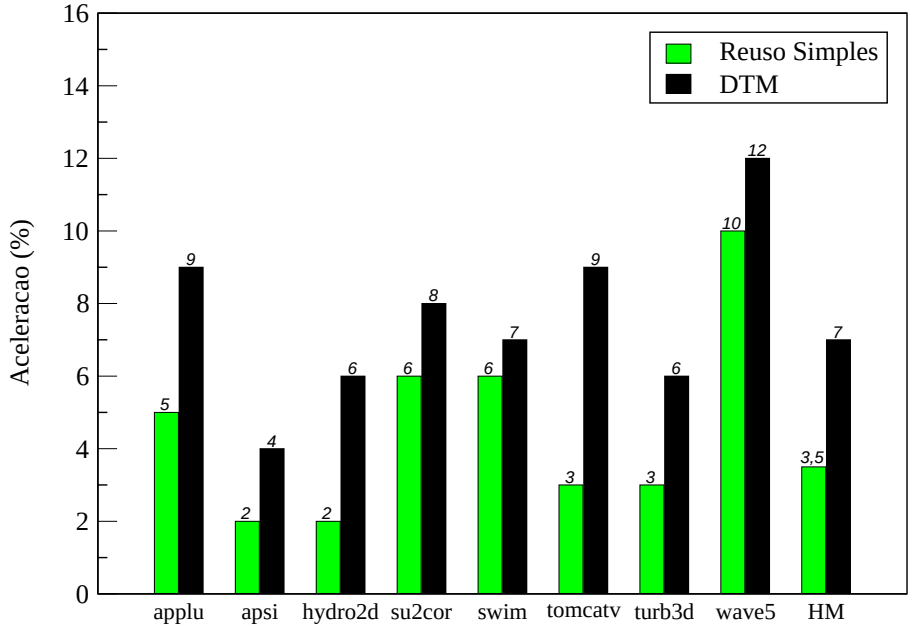


Figura 5.10: Comparação do percentual de ganhos de performance entre *Reuso Simples* e *DTM*, *SPECFp95*.

disponibilização antecipada de operandos em decorrência do reuso; etc....

5.2.4 Análise Comparativa

Esta subseção efetua uma análise comparativa do mecanismo *DTM* com outros mecanismos que exploram o reuso de computações redundantes. São feitas comparações entre o mecanismo *DTM* e os esquemas S_{n+d} [63] e *Block Reuse* [35] (todos o três esquemas usaram a mesma plataforma de simulação [16]). Nesta comparação, os esquemas apresentam-se equilibrados com relação a capacidade de armazenamento. Em ambas as medidas, o *DTM* apresenta *Memo_Table_T* com 512 entradas e *Memo_Table_G* com 4672 entradas (alguns bits foram incorporados à *Memo_Table_T* para marcar e identificar os registradores ativos do contexto de entrada e saída, reduzindo assim o número de entradas em *Memo_Table_G*), enquanto S_{n+d} apresenta 4096 entradas. Estes dois mecanismos apresentam a mesma capacidade de armazenamento em bits, ou seja, utilizam 784K bits em suas implementações. O *Block Reuse* apresenta uma configuração que exige uma quantidade maior de bits. Neste, serão considerados 1690 Kbits (2.16 vezes maior que S_{n+d} e *DTM*) para suportar uma *BHB* com 2048 entradas e configuração *Reg-In/Reg-Out/Mem-In/Mem-Out*

igual a 4/4/3/2, ou seja, contexto com 4 registradores de entrada, 4 registradores de saída, 3 operações de leitura à memória e 2 operações de escrita na memória. Os valores e os programas de teste utilizados para comparação com os mecanismos S_{n+d} e *Block Reuse*, foram obtidos de [65] e [35] respectivamente. Medidas não presentes na comparação são marcadas com o valor **0** e indicam que os respectivos programas não foram simulados. As médias harmônicas foram calculadas para os programas simulados que são comuns aos três mecanismos.

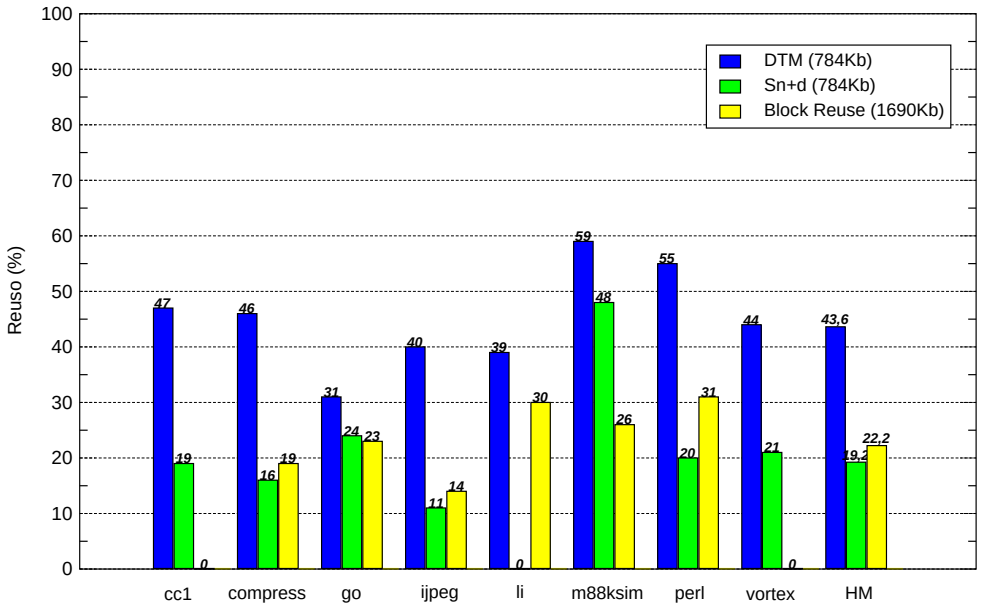


Figura 5.11: Comparação do reuso explorado *DTM*, S_{n+d}, *BHB*.

A Figura 5.11 compara o percentual de reuso explorado pelos três esquemas. Observa-se que o *DTM* reusa consideravelmente mais instruções que os demais esquemas comparados. O percentual de reuso identificado pelo *DTM* varia de 31% (*go*) a 59% (*m88ksim*) com média harmônica de 43.6%, enquanto para o S_{n+d} varia de 11% (*ijpeg*) a 48% (*m88ksim*) com média harmônica de 19.2% e para o *Block Reuse* varia de 14% (*ijpeg*) a 31% (*perl*) com média harmônica de 22.2%.

A Figura 5.12 compara o percentual de ganho de performance explorado pelos três esquemas. Observa-se que o *DTM* apresenta um considerável ganho de performance considerando os demais esquemas comparados. O ganho de performance identificado pelo *DTM* varia de 4% (*vortex*) a 21% (*compress*) com média harmônica de 10.2%, enquanto para o S_{n+d} varia de 3% (*compress*) a 10% (*vortex*) com média harmônica de 4.4% e para o *Block Reuse* varia de 1% (*ijpeg*) a 12% (*perl*) com média

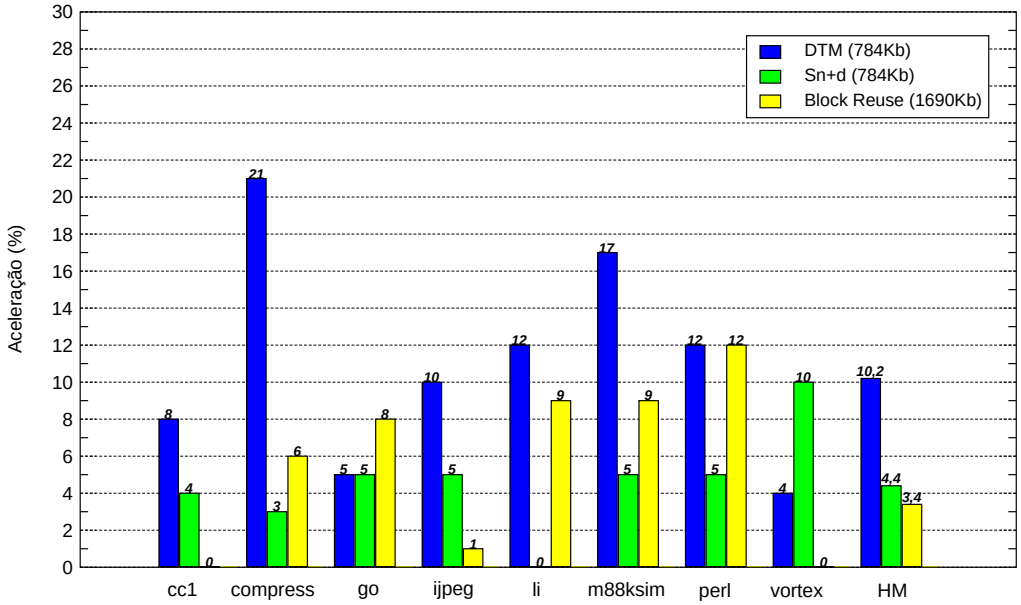


Figura 5.12: Comparação da aceleração de desempenho *DTM* X S_{n+d}, *BHB*.

harmônica de 3.4%.

5.3 Avaliações Experimentais

Nesta seção serão apresentadas várias medidas efetuadas para avaliar e caracterizar o mecanismo *DTM*.

Como já observado nos experimentos da seção 5.2, o percentual de reuso obtido não é o único fator determinante para obtenção de ganhos de performance. Diferentes instruções e traces redundantes promovem diferentes ganhos de performance. Estes ganhos de performance ainda dependem das circunstâncias em que se encontram os recursos do processador. Por exemplo, considerando um trace **T1** encapsulando três instruções e possuindo dentre elas duas instruções de desvio, este trace pode promover um bom ganho de performance, se o primeiro desvio encapsulado for predito incorretamente pelo processador. Neste caso, o reuso do trace **T1** corrigirá a predição (o segundo desvio é dependente do primeiro), e evitará a execução de caminhos especulativos. Entretanto, este mesmo trace **T1** pode em uma próxima oportunidade de reuso, promover um menor ganho de performance, caso o processador efetue corretamente as predições das instruções de desvio encapsuladas no trace. Em contrapartida, reusar um trace **T2** encapsulando 3 instruções que

são executadas completamente em um único ciclo pelo processador, certamente terá uma menor contribuição para obtenção de ganhos de performance se comparado com o trace **T1**. Estes exemplos procuram estabelecer que para o modelo de execução superescalar, o percentual de reuso identificado, contrariamente ao esperado intuitivamente, não é um parâmetro determinante do ganho de performance que pode-se obter através do reuso.

Determinar se uma dada composição de um trace redundante ou uma instrução redundante provocam muito ou pouco ganho de performance ao serem reusados, é uma tarefa extremamente complexa e depende de vários fatores correlacionados, tais como: número de instruções representadas no trace (tamanho do trace); quantidade de dependências verdadeiras que são encapsulados pelo trace; número de predições incorretas que são corrigidas pelo reuso do trace (evitando especulação); quantidade de desvios que são encapsulados pelo trace; quantidade de instruções fora do trace que são beneficiadas pelo reuso (disponibilização de operandos, antecipação de predições de desvio), etc.... Associados a estes fatores mencionados, existem ainda a ocorrência destes em um cenário que envolve considerações arquiteturais como: redução da pressão exercida por múltiplas requisições sobre as unidades funcionais; redução de acessos ao cache de instruções; crescimento artificial do tamanho do *buffer de emissão*; redução de requisições no *estágio de entrega*; etc.... Deve-se considerar ainda, que um mesmo trace pode apresentar diferentes efetividades (contribuições ao ganho de performance) ao ser reusado em diferentes períodos de processamento. Por exemplo, um trace que corrige uma predição incorreta quando reusado no ciclo k, produz uma efetividade maior do que o mesmo trace reusado em um ciclo $k+j$ que não corrige a predição efetuada, isto é, neste período de tempo o preditor alterou dinamicamente o seu histórico de predição e provocou uma queda na efetividade do trace reusado. Apesar destas dificuldades a serem consideradas, serão apresentados nas subseções subseqüentes, experimentos que medem determinadas características dos traces reusados. Estas medidas serão utilizadas para prover justificativas para alguns dos resultados obtidos.

5.3.1 Caracterização das instruções reusadas

Nesta subseção serão apresentados gráficos distinguindo a distribuição percentual por tipo de instruções reusadas pelo mecanismo *DTM*. Os gráficos apresentarão respectivamente, a distribuição de todas as instruções reusadas pelo mecanismo, das instruções reusadas isoladamente (provenientes de *Memo_Table_G*) e das instruções reusadas nos traces (provenientes apenas de *Memo_Table_T*). Na distribuição a ser apresentada, as instruções serão agrupadas por classes, e estas identificadas como instruções de desvio (condicionais, incondicionais e chamada e retorno de subrotinas), instruções aritméticas e lógicas, e instruções de cálculo de endereços. Apesar das instruções de cálculo de endereços serem instruções aritméticas, estas serão avaliadas isoladamente para caracterizar o limite imposto ao reuso, decorrente das operações de acesso à memória. As instruções de cálculo de endereços representam uma das operações efetuadas para o acesso à memória, e serão ainda subdivididas em instruções de cálculo de endereços para instruções de leitura, e escrita na memória. Esta subdivisão detalha pormenorizadamente os componentes que limitam a construção e o reuso de traces.

No gráfico da Figura 5.13 é apresentada a distribuição percentual por tipo de instrução, para todas as instruções que foram reusadas pelo mecanismo *DTM* (isoladamente e inclusas nos traces) para os programas do *SPECInt95*. Analisando os valores plotados, observa-se que 13,8%, 41.7%, 26.8% e 17.7% das instruções reusadas (média aritmética), correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória. Pode-se destacar que 44.5% das instruções reusadas representam instruções que efetuam o cálculo de endereços de acesso à memória.

No gráfico da Figura 5.14 é apresentada a distribuição percentual por tipo de instrução, para todas as instruções reusadas isoladamente (pertencentes à *Memo_Table_G*) pelo mecanismo *DTM*, para os programas do *SPECInt95*. Analisando os valores plotados, observa-se que 8.4%, 32.8%, 35% e 23.8% das instruções reusadas (média aritmética), correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória. Pode-se destacar que 58.8% das instruções reusadas representam instruções que efetuam o cálculo de endereços de acesso à memória.

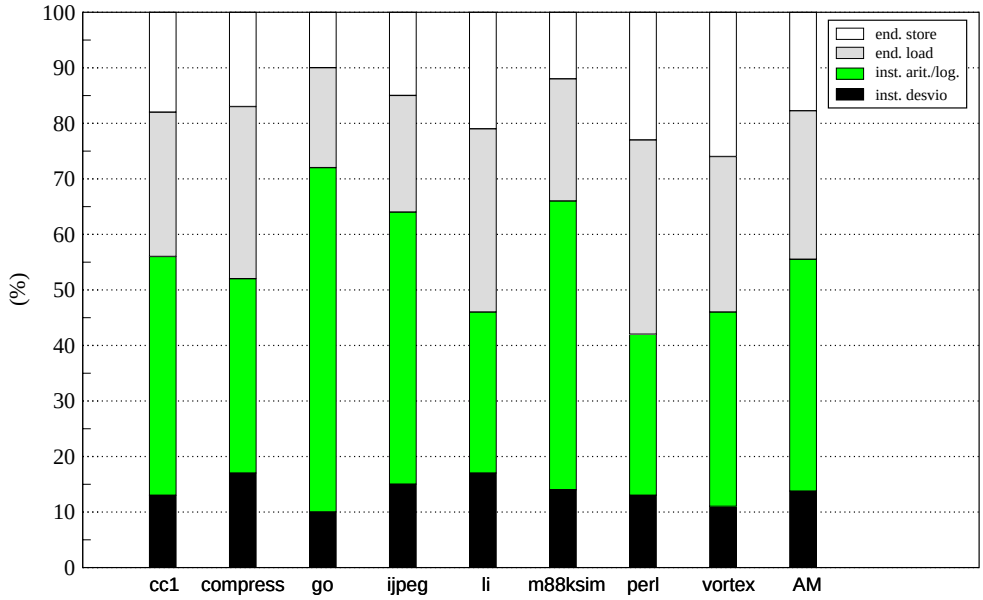


Figura 5.13: Distribuição do total de instruções reusadas (por tipo), *SPECInt95*.

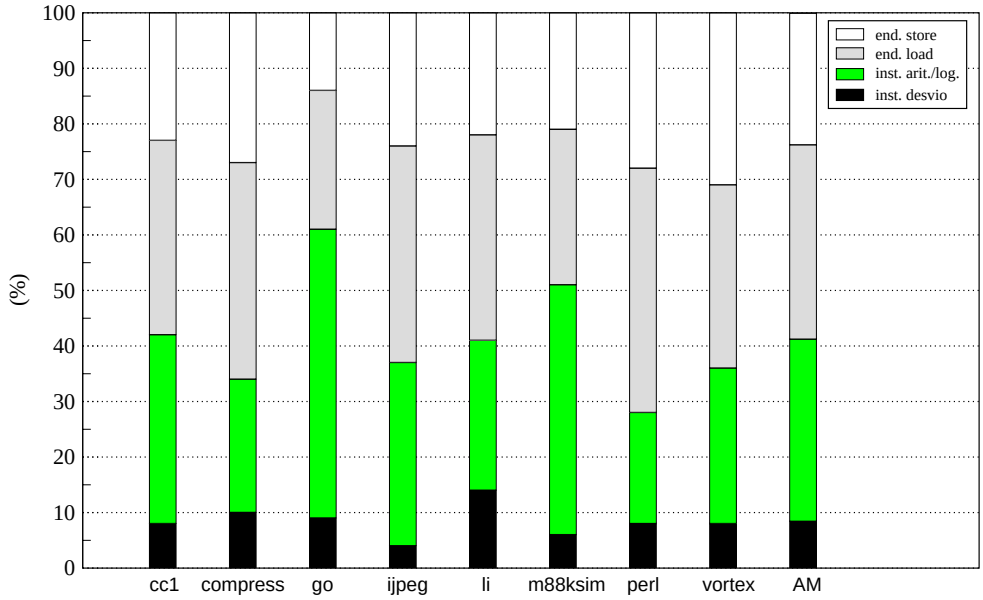


Figura 5.14: Distribuição das instruções reusadas isoladamente (por tipo), *SPECInt95*.

No gráfico da Figura 5.15 é apresentada a distribuição percentual por tipo de instrução, para todas as instruções representadas nos traces reusados (pertencentes à *Memo_Table_T*) pelo mecanismo *DTM*, para os programas do *SPECInt95*. Analisando os valores plotados, observa-se que 24.6%, 59.5%, 9.9% e 6% das instruções reusadas (média aritmética) correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória. Pode-se destacar que as instruções de desvio representam uma parcela razoável do total de instruções reusadas nos traces, e caracterizam traces encapsulando fluxos de execução. Observa-se também, que as instruções de cálculo de endereços de memória representam 15.9% do total de instruções reusadas nos traces. É importante ressaltar que o valor de 15.9% de instruções de cálculo de endereços de memória reusadas, representa uma grande barreira para a exploração de reuso, visto que estas instruções quando presentes em traces, sempre os finalizam. Considerando que os traces possuem em média 3 instruções (como será visto adiante), as instruções de cálculo de endereços de memória irão finalizar aproximadamente 42% dos traces (sem considerar as instruções de memória que não possuem o cálculo de endereço redundante), ou seja, a restrição de se reusar as operações de acesso à memória (redundância dos valores em memória) reduz a complexidade do mecanismo, porém pode restringir significativamente o número de instruções nos traces.

No gráfico da Figura 5.16 é apresentada a distribuição percentual por tipo de instrução, para todas as instruções reusadas pelo mecanismo *DTM* (isoladamente e as inclusas nos traces), para os programas do *SPECFp95*. Analisando os valores plotados, observa-se que 14%, 64.4%, 14.6% e 7% das instruções reusadas (média aritmética) correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória. Pode-se destacar que 14% das instruções reusadas representam instruções de desvio, e este percentual é praticamente idêntico ao mesmo percentual obtido nos programas do *SPECInt95*. Um caso interessante ocorre com o programa *applu*, onde nenhuma instrução de cálculo de endereço para escrita em memória é reusada.

No gráfico da Figura 5.17, é apresentada a distribuição percentual por tipo

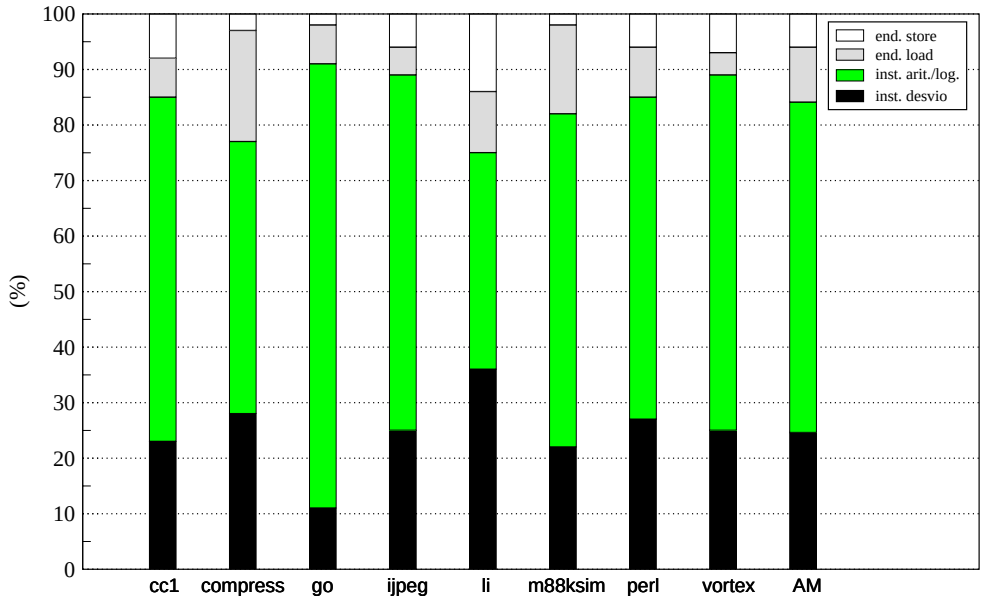


Figura 5.15: Distribuição das instruções reusadas em traces (por tipo), *SPECInt95*.

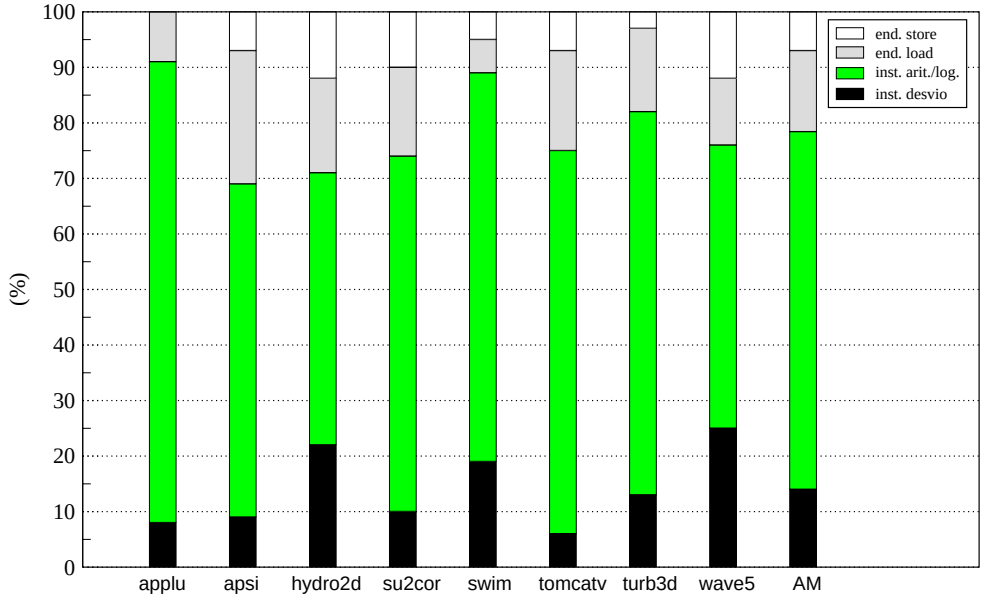


Figura 5.16: Distribuição do total das instr. reusadas (por tipo), *SPECFp95*.

de instrução, para todas as instruções reusadas isoladamente (pertencentes à *Memo_Table_G*) pelo mecanismo *DTM*, para os programas do *SPECFp95*. Analisando os valores plotados, observa-se que 13.3%, 49.4%, 25.3% e 12% das instruções reusadas (média aritmética) correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória.

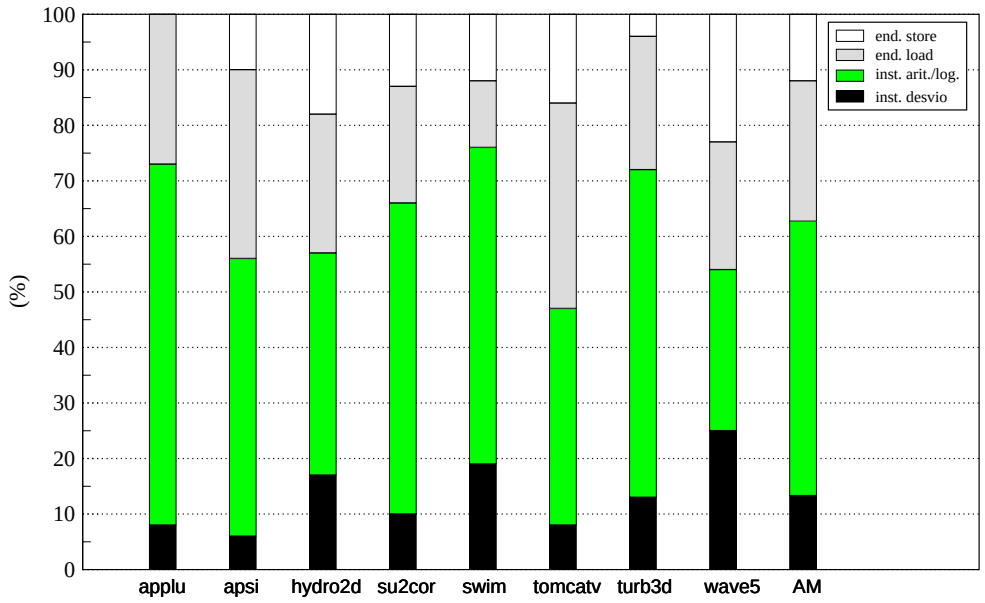


Figura 5.17: Distribuição das instr. reusadas isoladamente (por tipo), *SPECFp95*.

No gráfico da Figura 5.18 é apresentada a distribuição percentual por tipo de instrução, para as instruções representadas nos traces reusados (pertencentes à *Memo_Table_T*) pelo mecanismo *DTM*, para os programas do *SPECFp95*. Analisando os valores plotados, observa-se que 15.6%, 80%, 3% e 1.4% das instruções reusadas (média aritmética), correspondem respectivamente a instruções de desvio, aritméticas/lógicas, cálculo de endereço para leitura em memória e cálculo de endereço para escrita em memória. Pode-se destacar que 80% das instruções reusadas representam instruções que efetuam operações aritméticas/lógicas e que somente 4.4% das instruções reusadas são representadas por instruções de cálculo de endereços de memória. Contrariamente à análise feita para o *SPECInt95*, os traces obtidos nos programas do *SPECFp95* não sofrem restrições quanto ao percentual de 4.4% obtido para as instruções de cálculo de endereços de memória. Para este caso, as restrições impostas ao aumento do número de instruções em traces é limitado

pelas instruções de ponto flutuante, pois o mecanismo não suporta o tratamento destas.

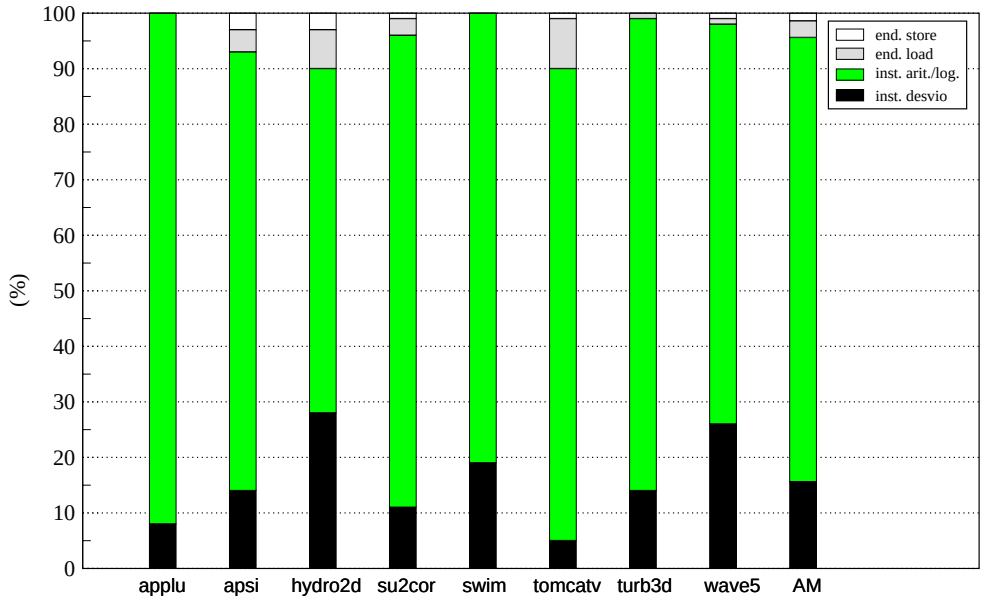


Figura 5.18: Distribuição das instruções reusadas em traces (por tipo), *SPECFp95*.

5.3.2 Avaliação dos contextos de entrada e saída dos traces reusados

As Figuras 5.19 e 5.20 apresentam uma distribuição percentual dos tamanhos (número de elementos) do contexto de entrada dos traces reusados. Esta medida determina uma condição arquitetural que define o número de elementos acessados e comparados em paralelo (por entrada em *Memo_Table_T*) para a verificar se um trace é redundante.

A partir dos resultados plotados na Figura 5.19, observa-se que para os programas do *SPECInt95*, os contextos de entrada estão distribuídos por valores (média aritmética) de 17.5% , 35.25%, 32.75%, 13% e 1.5%, para os respectivos números de elementos no contexto de entrada 0, 1, 2, 3 e 4. Este resultado revela que nenhum trace reusado possui um número de elementos no contexto de entrada maior que 4, e que 85.5% dos traces possuem contexto de entrada com menos de 3 elementos. Estes valores são razoáveis em termos de custos arquiteturais (tempo de acesso à *Memo_Table_T*), e indicam que os traces reusados podem suportar a inclusão de

instruções adicionais (aumentando seu tamanho) sem ultrapassar o limite de 4 elementos no contexto de entrada. Convém observar que contextos de entrada com nenhum elemento (tamanho 0), significa que o trace é composto por instruções que atribuem valores imediatos aos seus operandos de destino (ex: *add r3,r0,#45*).

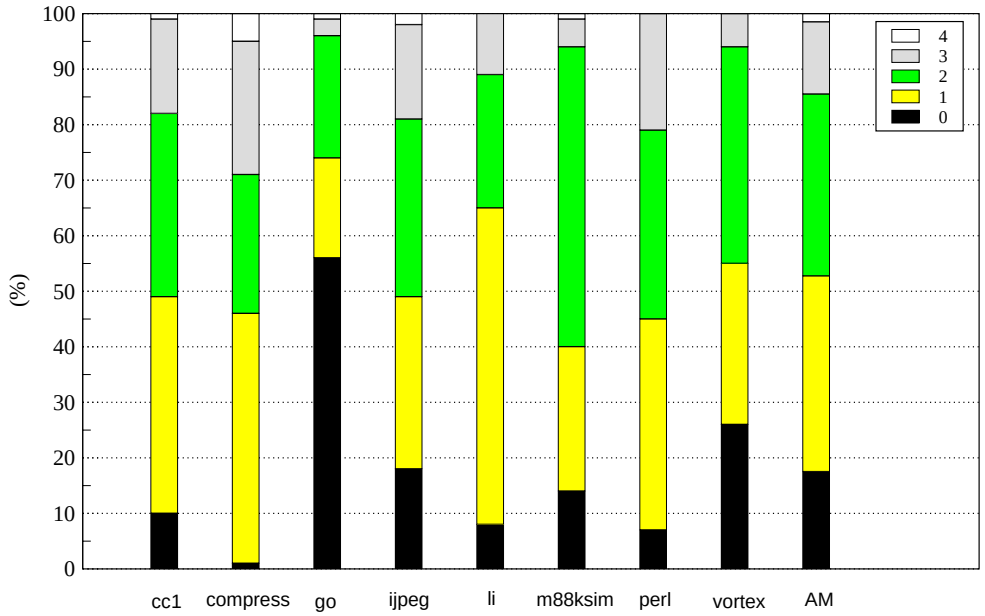


Figura 5.19: Distribuição percentual do número de elementos no contexto de entrada para os traces reusados, *SPECInt95*.

A Figura 5.20 apresenta a avaliação do número de elementos no contexto de entrada para os traces reusados pelo *DTM*, considerando os programas do *SPECFp95*. Seguindo a mesma linha de explanação efetuada para os programas que manipulam valores inteiros, nos programas do *SPECFp95*, os contextos de entrada estão distribuídos por valores (média aritmética) de 20.9% , 24.7%, 35.5%, 12.4%, 3.8% e 2.7%, para os respectivos números de elementos no contexto de entrada 0, 1, 2, 3, 4 e 5. Este resultado revela um comportamento semelhante aos resultados obtidos para os programas do *SPECInt95*. A identificação de traces com contexto de entrada com 5 elementos, praticamente não insere nenhuma problema, visto que estes não são representativos na distribuição. Observa-se também para este caso, que 81.1% dos traces reusados, possuem contexto de entrada com menos de 3 elementos.

As Figuras 5.21 e 5.22 apresentam uma distribuição percentual dos tamanhos (número de elementos) do contexto de saída dos traces reusados. Esta medida determina uma condição arquitetural que define o número de instruções que serão

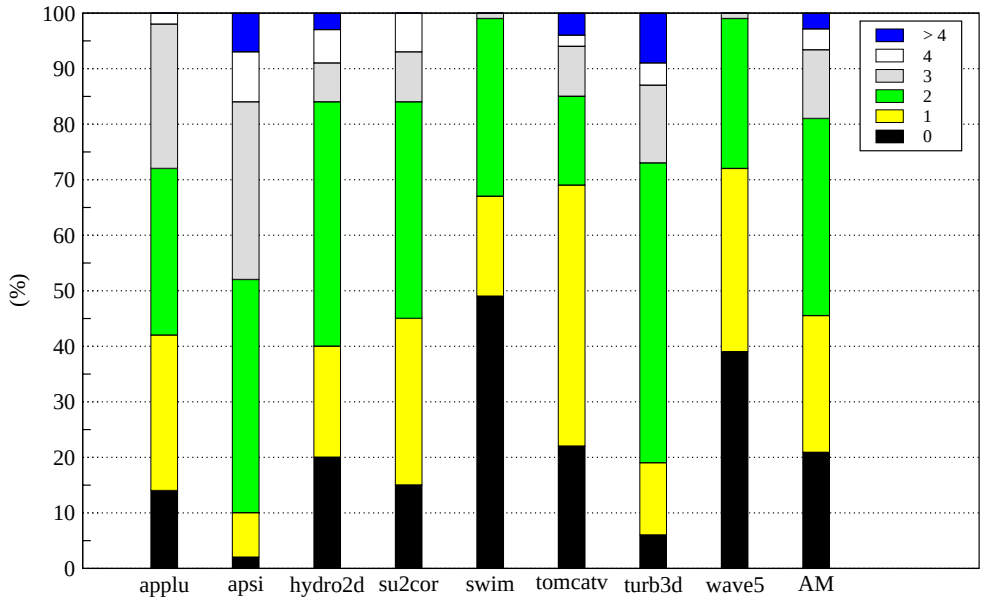


Figura 5.20: Distribuição percentual do número de elementos no contexto de entrada para os traces reusados, *SPECFp95*.

despachadas para atualizar o contexto do arquivo de registradores após o reuso de um trace. O contexto de saída será composto por registradores e seus respectivos valores. Estes representam o estado do processador que deverá estar ativo após o reuso de um trace. Quando um trace for reusado, o contexto de saída será despachado (a arquitetura provê o *buffer de emissão* e o *buffer de reordenação* em uma mesma estrutura), porém não executado. O despacho se faz necessário para que o processador substrato continue a suportar interrupções precisas [60], e o contexto de saída ficará aguardando a sua retirada do *buffer de emissão/reordenação* pelo *estágio de entrega*. É importante notar que o reuso de traces provoca um aumento artificial do *buffer de emissão*, visto que o contexto de saída representa muitas das vezes, uma fração do total de instruções representadas no trace (no pior caso, o mesmo número de instruções).

A partir dos resultados plotados na Figura 5.21, observa-se que para os programas do *SPECInt95*, os contextos de saída estão distribuídos por valores (média aritmética) de 3.6% , 41.2%, 37.9%, 15.2% e 2.1%, para os respectivos números de elementos no contexto de saída 0, 1, 2, 3 e 4. Este resultado revela que nenhum trace reusado possui um número de elementos no contexto de saída maior que 4, e que 79.1% dos traces possuem contexto de saída concentrados entre 1 e 2 elementos.

A constatação de que 41.2% dos traces possuem apenas um elemento no contexto de saída, equivale a afirmar que para o pior caso (os traces considerados possuem no mínimo 2 instruções representadas), o trace reusado ocupa somente uma posição no *buffer de emissão/reordenação* ao invés de 2 (quando a execução é normal). Esta consideração reafirma o aumento artificial do *buffer de emissão/reordenação*, pois as instruções reusadas pelo trace (seu contexto de saída) irão ocupar um número menor de entradas no *buffer de emissão/reordenação*, liberando assim um número maior de entradas para serem alocadas para outras instruções, e possibilitando uma maior exposição de *ILP* [62, 61]. Raciocínio análogo se aplica ao número de instruções entregues por ciclo pelo *estágio de entrega*, sendo sua atividade reduzida e conseqüentemente reduzida também, a pressão exercida para atualizações no arquivo de registradores. Convém observar que contextos de saída com nenhum elemento (tamanho 0), significa que o trace é composto somente por instruções de desvio (ex: trace *200 bneq r1,r2,#64*).

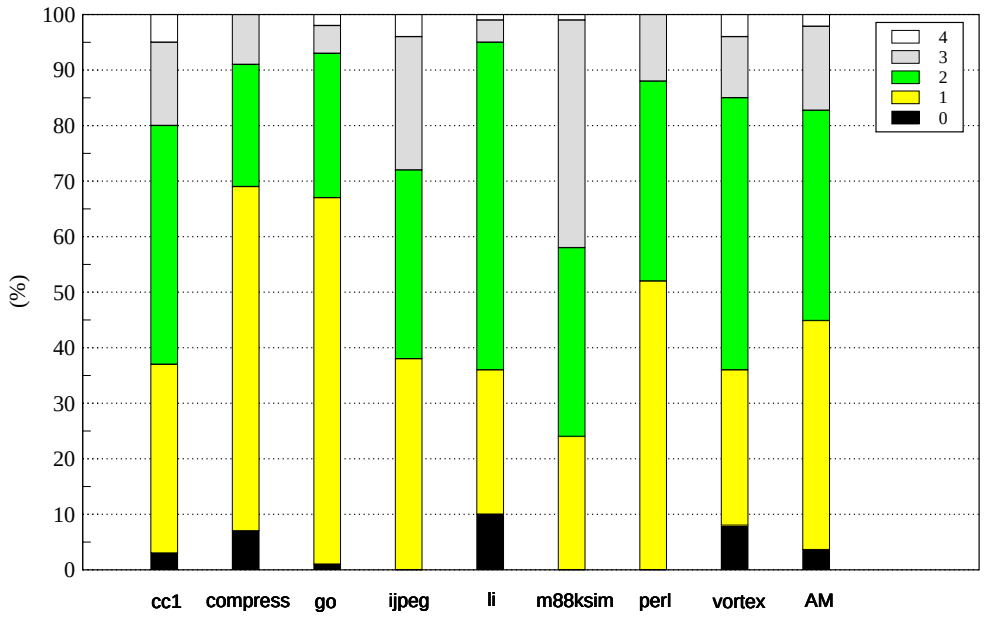


Figura 5.21: Distribuição percentual do número de elementos no contexto de saída para os traces reusados, *SPECInt95*.

A partir dos resultados plotados na Figura 5.22, observa-se que para os programas do *SPECFp95*, os contextos de saída estão distribuídos por valores (média aritmética) de 0.5% , 53.6%, 32.5%, 9.8%, 2.7% e 0.9%, para os respectivos números de elementos no contexto de saída 0, 1, 2, 3, 4 e 5. Este resultado revela que 86.1%

dos traces possuem contexto de saída concentrados entre 1 e 2 elementos, e são análogas às considerações feitas para o *SPECInt95*.

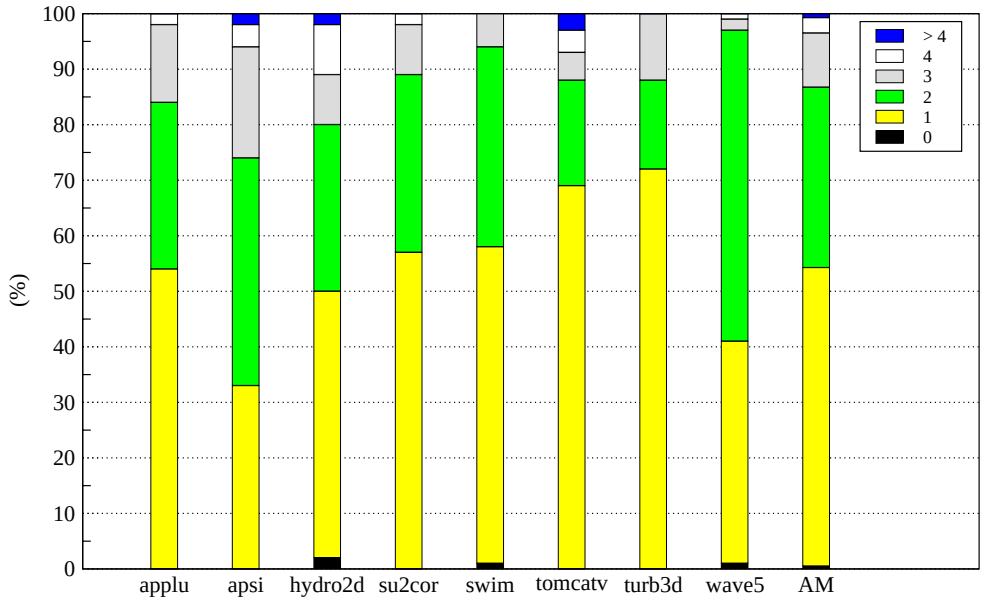


Figura 5.22: Distribuição percentual do número de elementos no contexto de saída para os traces reusados, *SPECFp95*.

5.3.3 Avaliação do número de instruções inclusas nos traces reusados

O número de instruções encapsuladas em um trace redundante representa o número de instruções que serão reusadas de uma só vez, ou ainda, o número de instruções que não serão executadas cada vez que um trace for reusado. A medida a seguir possui como objetivo, expor percentualmente os traces com diferentes números de instruções para cada programa avaliado. Para identificar o número médio de instruções encapsuladas nos traces reusados será considerada a seguinte métrica:

pt_n - Percentual de traces reusados com n instruções.

nmit - Número médio de instruções nos traces reusados, $\frac{1}{100}(\sum_{i=2}^{n} pt_i * i)$

A Figura 5.23 apresenta a distribuição percentual do número de instruções (ou tamanho) representadas nos traces. Considerando o *SPECInt95*, observa-se que os traces possuem tamanhos variando de 2 a 5 instruções (para alguns programas

foram identificados traces com até 16 instruções). O valor 3.0, representa o número médio de instruções nos traces para o conjunto de programas do *SPECInt95*. Os programas *go* e *m88ksim* apresentaram respectivamente o menor (2.5) e o maior (3.8) *nmit*. Analisando a distribuição (média aritmética), observou-se que os traces de tamanho 2 , 3 , 4 , 5 e >5 correspondem respectivamente a 34.2%, 41.8%, 11%, 9% e 4% do total dos traces reusados.

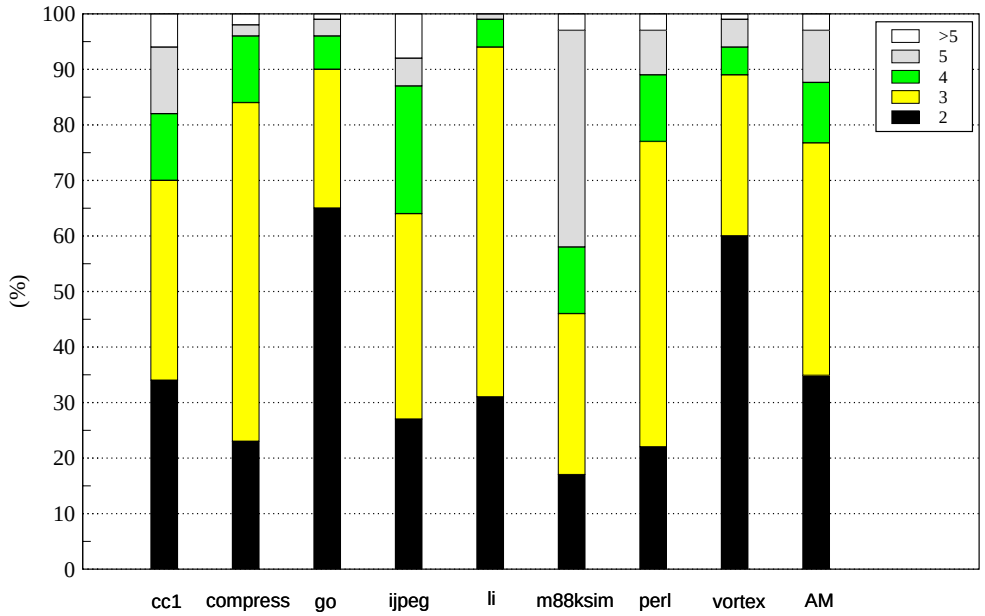


Figura 5.23: Distribuição percentual do número de instruções nos traces reusados, *SPECInt95*.

Na Figura 5.24, considerando o *SPECFp95*, observa-se que os traces possuem tamanhos variando de 2 a 6 instruções. O valor 2.8 representa o número médio de instruções nos traces para o conjunto de programas do *SPECFp95*. Os programas *swim* e *tomcatv* apresentaram respectivamente o menor (2.3) e o maior (3.5) *nmit*. Analisando a distribuição (média aritmética), observou-se que os traces de tamanho 2, 3, 4, 5, 6 e >6 correspondem respectivamente a 50%, 30%, 12%, 3%, 2.5% e 2.5% do total dos traces reusados.

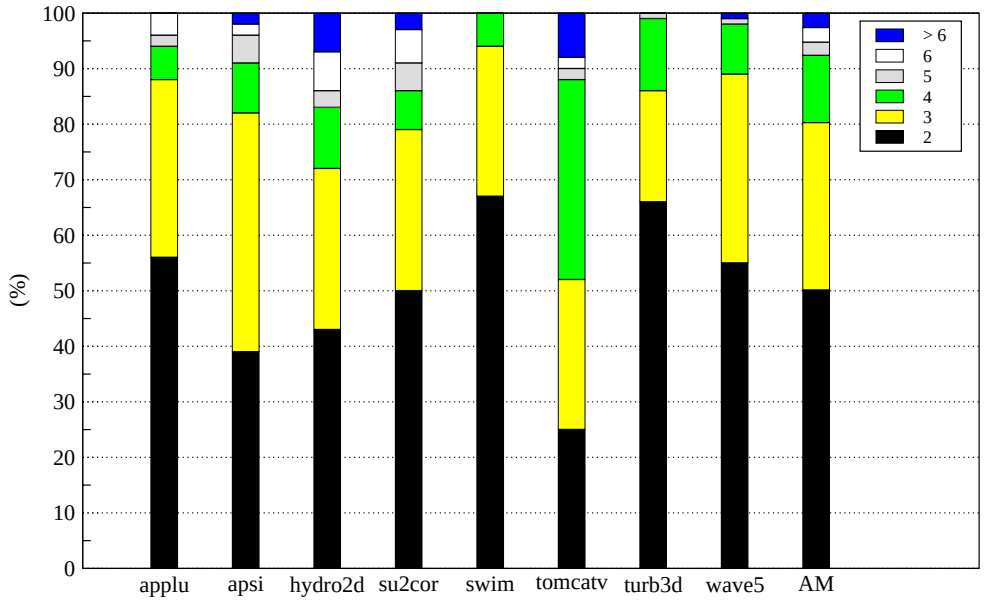


Figura 5.24: Distribuição percentual do número de instruções nos traces reusados, *SPECFp95*.

5.3.4 Quantidade de desvios encapsulados nos traces reusados

A avaliação a seguir, caracteriza a capacidade de se reusar seqüências de instruções ultrapassando as fronteiras de blocos básicos, ou seja, o encapsulamento de fluxos de execução.

A Figura 5.25 apresenta uma avaliação considerando os programas do *SPECInt95*, para estes, observou-se que os traces com 0, 1, 2, 3 e 4 desvios encapsulados, correspondem respectivamente a 38.6%, 40.5%, 18%, 1.5% e 1.4% do total de traces reusados. A partir das medidas obtidas, observa-se que 61.4% dos traces reusados possuem pelo menos uma instrução de desvio. O programa *m88ksim* apresentou 49% de seus traces com 2 ou mais desvios encapsulados, enquanto o programa *li* apresentou 61% de seus traces sem nenhum desvio encapsulado. Para estes mesmos programas foram obtidas anteriormente na comparação *DTM X Reuso Simples* na subseção 5.2.3, as maiores (*m88ksim*) e menores (*li*) diferenças entre os ganhos de performance da configuração *DTM* e *Reuso Simples*. Esta consideração identifica uma das características favoráveis ao reuso de traces.

Efetuando a mesma análise para os programas do *SPECFp95*, na Figura 5.26,

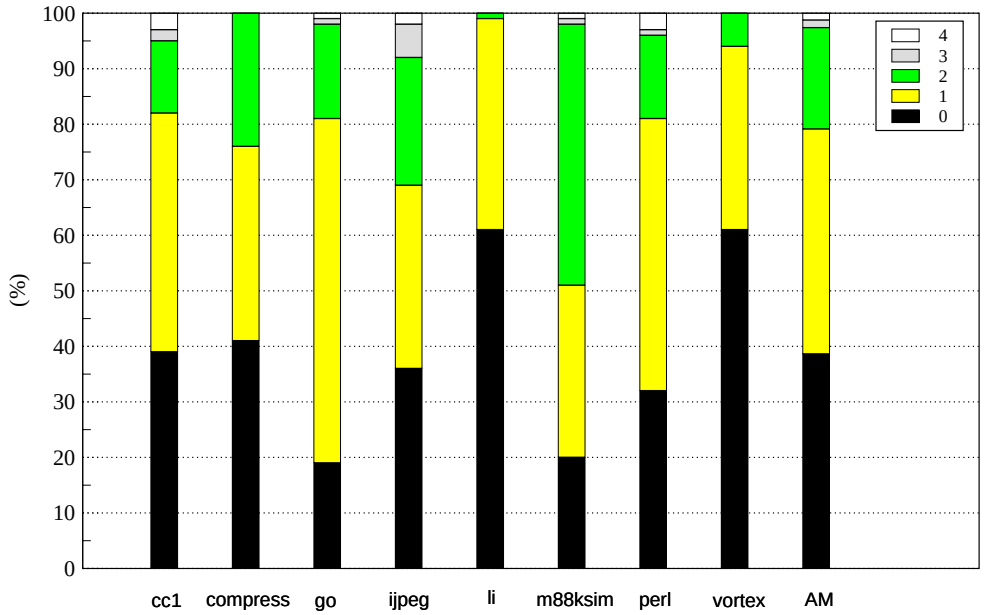


Figura 5.25: Distribuição percentual do número de desvios inclusos nos traces reusados, *SPECInt95*.

observa-se que os traces com 0, 1, 2, 3 e 4 desvios encapsulados, correspondem respectivamente a 26%, 52%, 14%, 5.2% e 2.8% do total de traces reusados, e que 74% dos traces reusados possuem pelo menos uma instrução de desvio. O programa *tomcatv* apresentou 56% de seus traces com 2 ou mais desvios encapsulados, enquanto o programa *apsi* apresentou 47% de seus traces sem nenhum desvio encapsulado. Para o programa *tomcatv* foi obtido anteriormente na comparação *DTM X Reuso Simples*, a maior diferença entre os ganhos de performance da configuração *DTM* e *Reuso Simples*.

5.3.5 Distribuição percentual do comprimento dos caminhos críticos encapsulados nos traces reusados

A avaliação desta subseção procura identificar nos traces reusados, os percentuais de ocorrência e o comprimento dos *caminhos críticos* de execução [7, 40, 1]. Nos experimentos realizados, os traces reusados foram classificados quanto ao comprimento do *caminho crítico*. O *caminho crítico* incluso em um trace reusado irá considerar apenas as dependências verdadeiras entre as instruções do trace, ou seja, somente a serialização imposta por estas, será considerada como obstáculo que restringe o

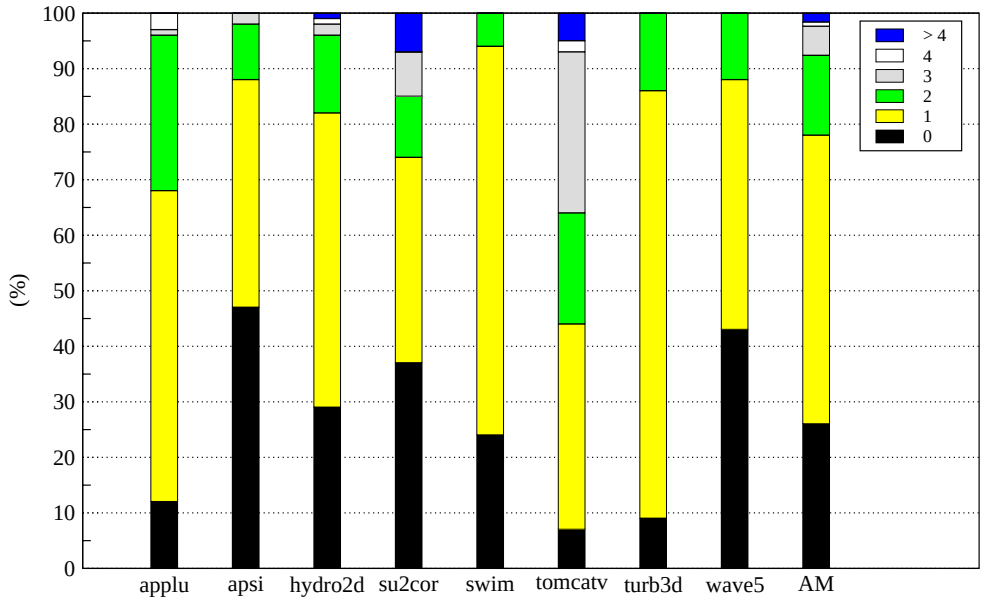


Figura 5.26: Distribuição percentual do número de desvios inclusos nos traces reusados, *SPECFp95*.

paralelismo disponível [5].

As Figuras 5.27(a) e 5.27(b) esboçam a obtenção do comprimento dos *caminhos críticos*, desconsiderando os parâmetros arquiteturais, os efeitos das dependência de controle e recursos funcionais. É importante ressaltar que um trace ao ser reusado promove o *collapsing* de toda a seqüência de instruções, independente do comprimento do *caminho crítico* [58, 27]. Para o exemplo da Figura 5.27(a), o caminho crítico é determinado pelas dependências verdadeiras existentes entre as instruções nos endereços *100*, *104* e *108*. Enquanto para o exemplo da Figura 5.27(b), o caminho crítico é determinado pelas instruções nos endereços *100* e *404*.

Na Figura 5.28 é apresentada para os programas do *SPECInt95*, a distribuição percentual dos traces reusados, considerando o comprimento do *caminho crítico* encapsulado. Analisando os valores plotados, observa-se em média (aritmética) que: 37%, 50%, 11% e 2% dos traces reusados, encapsulam seqüências de instruções dinâmicas possuindo os *caminhos críticos* com comprimentos 1, 2, 3 e >3 respectivamente. A partir desta exposição, pode-se concluir que 63% dos traces reusados estão aptos a efetuar *collapsing* de dependências verdadeiras. Observando ainda o gráfico plotado, verifica-se que os programas *li* e *go*, possuem respectivamente 89% e 23% (maior e menor percentual respectivamente) dos traces reusados efetuando

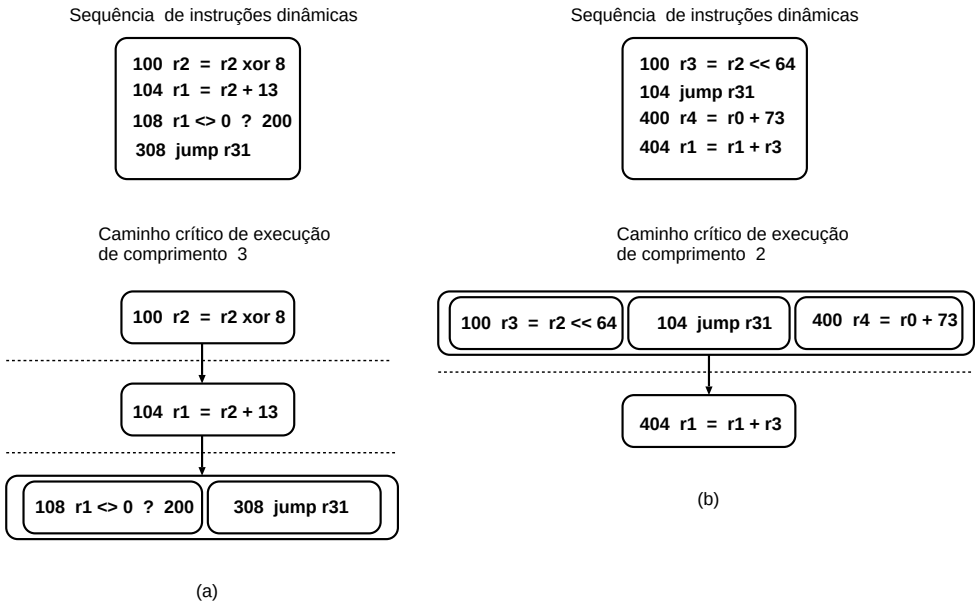


Figura 5.27: Exemplos de traces e respectivos grafos representando o comprimento do *caminho crítico*.

collapsing. Analisando o programa *compress*, observa-se que 55% dos traces reusados efetuam *collapsing* e que 29% de seus traces reusados efetuam *collapsing* em *caminhos críticos* de comprimento maior que 3.

Na Figura 5.29 é apresentada para o *SPECFp95*, a distribuição percentual dos traces reusados considerando o comprimento do *caminho crítico* encapsulado. Analisando os valores plotados, observa-se em média (aritmética) que: 64%, 28.7%, 6.5% e 0.8% dos traces reusados, encapsulam seqüências de instruções dinâmicas possuindo os *caminhos críticos* com comprimentos 1, 2, 3 e >3 respectivamente. A partir desta exposição, pode-se concluir que apenas 36% dos traces reusados estão aptos a efetuar *"collapsing"* de dependências verdadeiras. Observando ainda o gráfico plotado, verifica-se que para o programa *hydro2d*, 73% dos traces reusados efetuam *collapsing* e que 16% de seus traces reusados efetuam *collapsing* em *caminhos críticos* de comprimento maior que 3. Este resultado era esperado para estes programas, visto que nestes, as dependências verdadeiras são acentuadas entre as operações de ponto flutuante, e estas não são consideradas pelo mecanismo *DTM*.

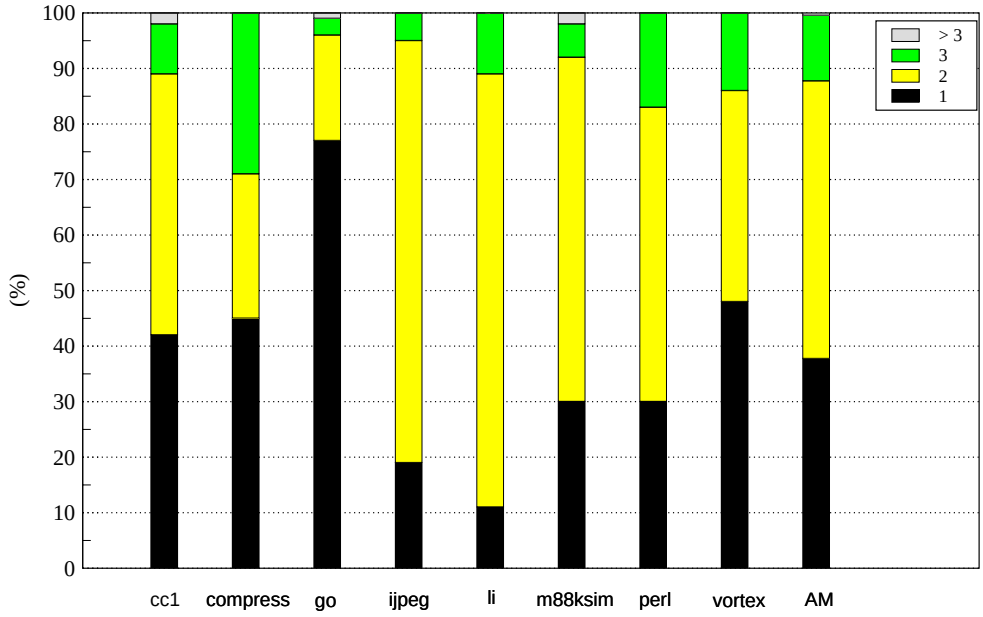


Figura 5.28: Distribuição percentual dos traces reusados no *SPECInt95*, considerando o comprimento do *caminho crítico* encapsulado.

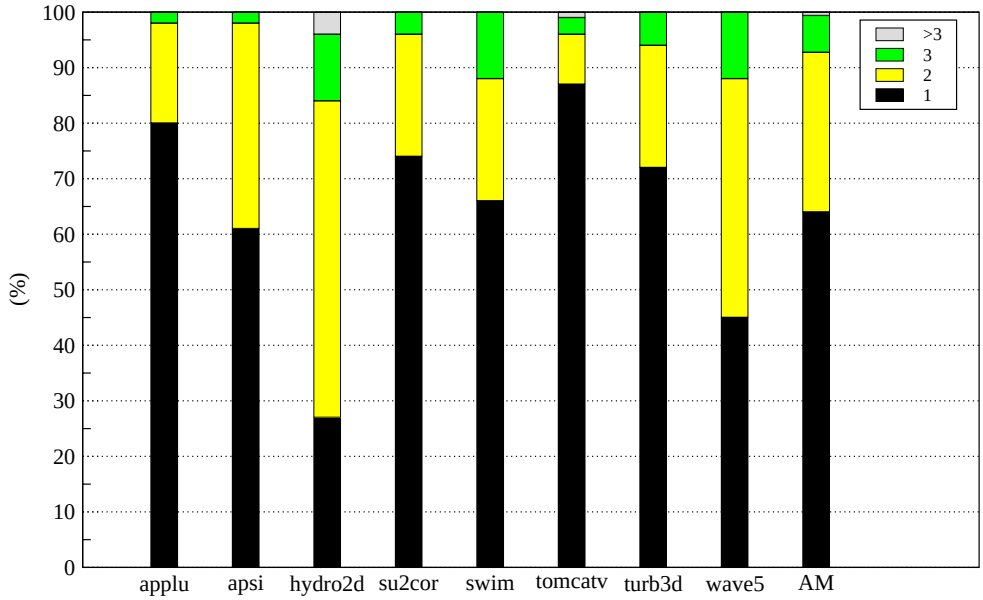


Figura 5.29: Distribuição percentual dos traces reusados no *SPECFp95*, considerando o comprimento do *caminho crítico* encapsulado.

5.3.6 Quantificação dos recursos requeridos pelo mecanismo *DTM*

Nesta subseção serão apresentadas algumas avaliações considerando os recursos arquiteturais extras que são requeridos ao processador superescalar substrato, quando da incorporação do mecanismo *DTM*.

Número de portas de leitura requisitadas pelo mecanismo *DTM*

Uma importante consideração arquitetural e já mencionada no capítulo 4, refere-se ao possível aumento do número de requisições efetuadas ao arquivo de registradores, quando da incorporação do *DTM* à arquitetura substrato. Este aumento de requisições decorre do fato de que além das requisições convencionais efetuadas pelas instruções a serem despachadas, novas requisições são efetuadas pelos traces a serem testados quanto ao reuso. O somatório destas requisições poderia requerer um aumento do número de portas de leitura no arquivo de registradores, adicionando custo e uma maior latência de acesso ao mesmo.

As medidas efetuadas verificaram a cada ciclo, o número de leituras requisitadas ao arquivo de registradores. Nas figuras a serem apresentadas, as barras com o rótulo **A** identificam o número de portas requisitadas ao arquivo de registradores quando é considerado somente o processador substrato, enquanto as barras rotuladas com o rótulo **B** identificam o número de leituras requisitadas pelo processador substrato incorporando o mecanismo *DTM*. Estas, incluem as requisições feitas pelas instruções a serem despachadas (comuns ao processador substrato), adicionada às requisições feitas pelos traces candidatos a reuso (registradores do contexto de entrada). Esta identificação é usada para plotar os gráficos das Figuras 5.30 e 5.31. Observando os gráficos plotados, verifica-se para o *SPEC95*, que nenhuma requisição conjunta ultrapassou o número de portas de leitura do processador substrato (8 portas) e que as requisições adicionais efetuadas pelos traces não adicionam uma pressão significativa no arquivo de registradores. Considerando o número de portas de saída, nenhuma porta adicional será requerida ao arquivo de registradores, visto que o contexto de saída de um trace reusado é escalonado no *buffer de emissão/reordenação* (sendo marcado como reusado e não executado), e retirado normalmente pelo estágio de entrega (limitado aos recursos disponíveis no processador substrato). É importante

mencionar que para todas as medidas, foram consideradas todas as requisições ao arquivo de registradores para todos os traces candidatos à reuso, ou seja, os reusados e os não reusados.

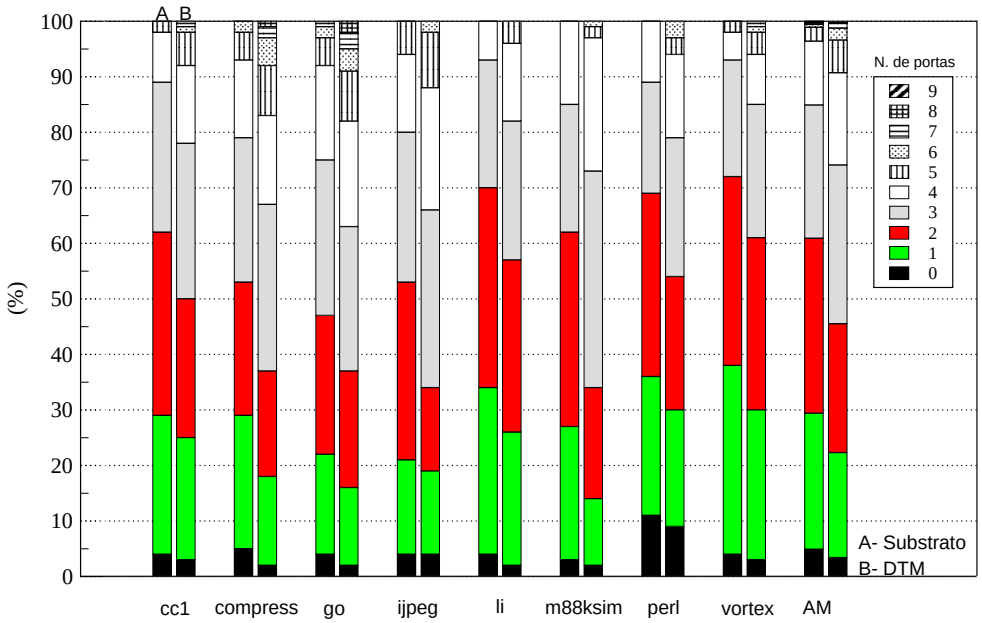


Figura 5.30: Número de portas de leitura do arquivo de registradores requisitadas pelo processador substrato e pelo processador substrato incorporando o mecanismo *DTM*, *SPECInt95*.

Número de portas de leitura e escrita necessárias para acesso à *Memo_Table_T*

Esta avaliação determina o número de portas de leitura e escrita requeridas pelo mecanismo *DTM* para suportar as requisições feitas à *Memo_Table_T*. Analisando inicialmente o número de portas de escrita, é imediato concluir que *Memo_Table_T* necessita de apenas uma porta de escrita para efetuar a inclusão de traces em *Memo_Table_T*, pois traces são inseridos quando finalizada sua construção, e esta só ocorre em diferentes ciclos. O número de portas de leitura é determinado, avaliando-se em cada ciclo em que novas instruções são inseridas nos estágios de busca e/ou decodificação, o número de requisições feitas à *Memo_Table_T* pelos estágios **DS1** e **DS2** descritos em 4.1. A coleta destas requisições considera os acessos simultâneos que podem ser efetuadas por **DS1** e **DS2**, ou seja, enquanto traces são pré-selecionados pelo campo *pc* no estágio **DS1**, traces previamente selecionados

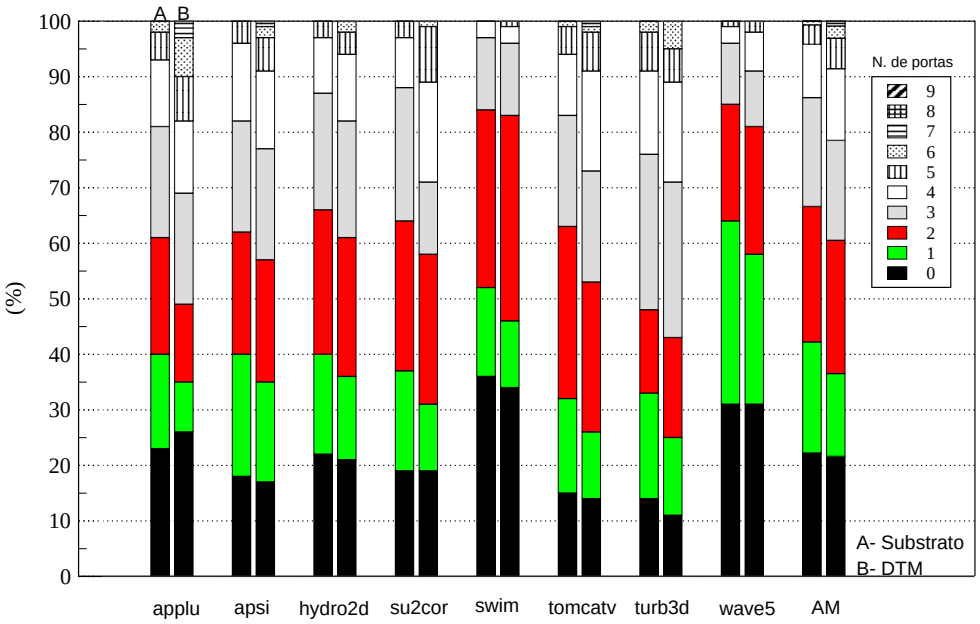


Figura 5.31: Número de portas de leitura do arquivo de registradores requisitadas pelo processador substrato e pelo processador substrato incorporando o mecanismo *DTM*, *SPECFp95*.

(pelo estágio **DS1**) podem ser avaliados quanto a sua redundância (comparação de contexto de entrada) em **DS2**. É importante mencionar, que foram consideradas todas as requisições à *Memo_Table_T*, para todos os traces candidatos à reuso, ou seja, os reusados e os não reusados.

As Figuras 5.32 e 5.33 plotam os valores obtidos para os programas do *SPEC95*. Os valores 0, 1, 2, 3, e 4 apresentados na legenda de ambos os gráficos, identificam o número de acessos efetuados à *Memo_Table_T*. Para os programas do *SPECInt95* a distribuição percentual em média aritmética destes acessos foram de 34.6%, 35.6%, 20.4%, 8% e 1.4% respectivamente. Enquanto para os programas do *SPECFp95* a distribuição percentual em média aritmética destes acessos foram de 48.9%, 33.7%, 14.8%, 2.4% e 0.2% respectivamente. A partir dos valores apresentados, pode-se concluir que uma *Memo_Table_T* com 4 portas de leitura satisfaz completamente todas as requisições para a identificação de traces redundantes. Entretanto, observa-se também, que em cada ciclo em que novas instruções são inseridas nos estágios de busca e/ou decodificação:

• Em 34.6% e 48.9% destes ciclos, para os programas do *SPECInt95* e *SPECFp95* respectivamente, a *Memo_Table_T* não é acessada para leitura.

• Em 90.6% e 97.6% destes ciclos, para os programas do *SPECInt95* e *SPECFp95* respectivamente, uma *Memo_Table_T* com 2 portas de leitura satisfaria todas as requisições de acesso para leitura.

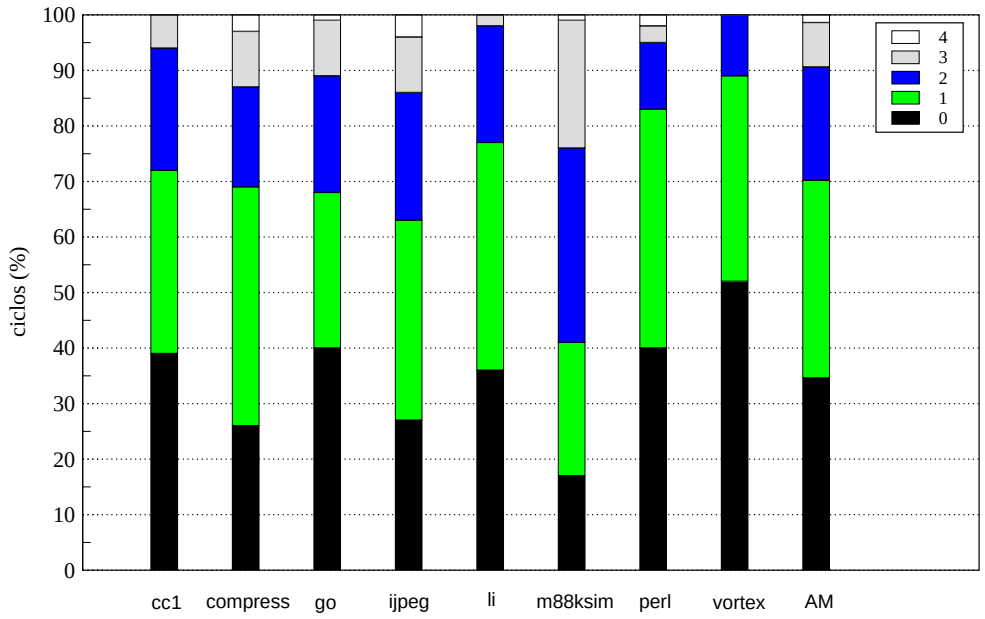


Figura 5.32: Distribuição percentual de requisições de leitura à *Memo_Table_T* em cada ciclo em que novas instruções são inseridas nos estágios de busca e/ou decodificação, *SPECInt95*.

5.3.7 Avaliação dos efeitos causados pelos caches e preditor de desvios no mecanismo *DTM*

Nesta subseção será avaliado o desempenho do mecanismo *DTM*, considerando o efeito dos caches e do preditor de desvios inclusos no processador superescalar substrato que o incorpora. As medidas efetuadas possuem como objetivo principal, identificar como alguns dos elementos básicos de configuração atuam positiva ou negativamente no desempenho do *DTM*, afetando o reuso e os ganhos de performance obtidos.

O reuso explorado pelo mecanismo *DTM* provoca ganhos de performance que podem ser oriundos de diferentes fontes, entre elas, pode-se destacar a correção de desvios preditos incorretamente. Esta característica impede a execução de instruções especulativas em caminhos incorretos e conseqüentemente reduz a demanda

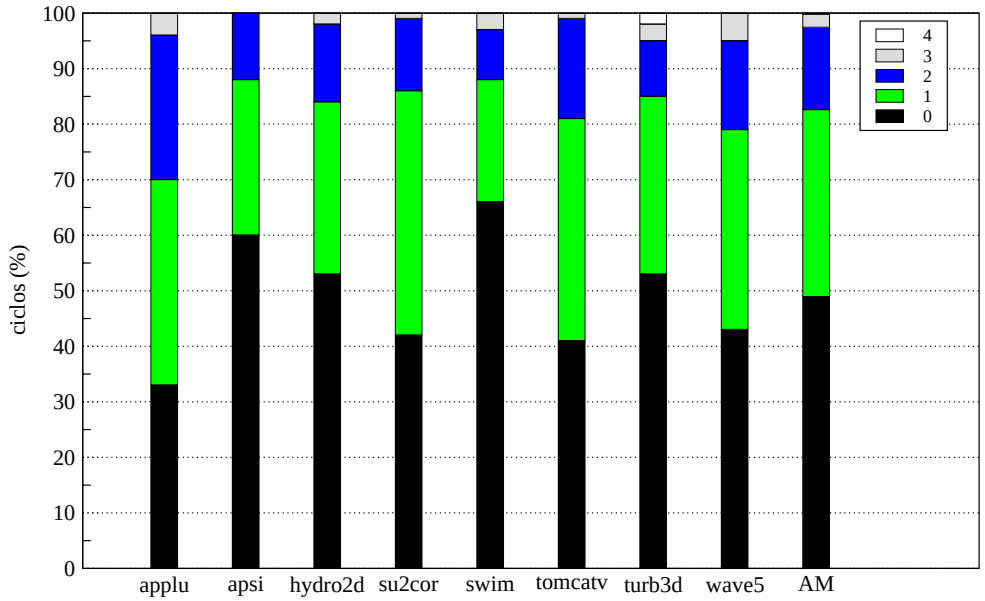


Figura 5.33: Distribuição percentual de requisições de leitura à *Memo_Table_T* em cada ciclo em que novas instruções são inseridas nos estágios de busca e/ou decodificação, *SPECFp95*.

por recursos. As tabelas 5.8 e 5.9, apresentam para o *SPECInt95* e *SPECFp95* respectivamente, o percentual de:

- *direção* - Direções de desvios condicionais que foram preditas incorretamente e corrigidos pelo *DTM*;
- *JR* - Predições de desvios incondicionais indiretos que foram corrigidas pelo *DTM* (endereço alvo em registrador);
- *I-Cache* - Reduções de acessos feitos ao cache de instruções *L1*. Este foram evitados em decorrência da não execução de caminhos especulativamente incorretos, ou evitados quando um trace é reusado e o mesmo dispensa a busca de instruções (pode ocorrer concomitantemente com um acesso sem sucesso ao cache de instruções *L1*);
- *loads em D-cache* - Reduções de acessos feitos ao cache de dados *L1* em decorrência da não execução de caminhos especulativamente incorretos.

Os valores asociados à *direção* e *JR*, apresentam os respectivos percentuais de desvios (sobre o total de desvios preditos) que foram preditos incorretamente e corrigidos pelo *DTM*. Enquanto os valores associados à *I-Cache* e *loads em D-cache*, apresentam os respectivos valores percentuais de acessos aos respectivos caches (so-

bre o total de acessos) que foram evitados em decorrência do reuso explorado pelo *DTM*.

Tabela 5.8: Efeitos provocados pelo reuso explorado, *SPECInt95*.

Benchmark	**direção %**	**JR %**	**I-Cache %**	**loads em D-cache %**
cc1	7.9	23.3	9	5
compress	7	12.8	14	5
go	10.6	19.2	5	2.5
ijpeg	5.1	0	10	3
li	4.7	46	8.5	3.3
m88ksim	4.3	28.3	15	3
perl	5.2	45.3	10	2.5
vortex	1.3	0.5	2.5	0.5
média arit.	5.8	22	9.2	3.1

Tabela 5.9: Efeitos provocados pelo reuso explorado, *SPECFp95*.

Benchmark	**direção %**	**JR %**	**I-Cache %**	**loads em D-Cache %**
applu	24.6	10.5	12	5
apsi	4.6	33	5	1.5
hydro2d	1.5	31	7.5	1.5
su2cor	1.5	75.7	6	2
swim	0	97.3	7.5	2.3
tomcatv	2.7	39.5	12.5	1.5
turb3d	7.6	66.7	5.5	3
wave5	0	83.6	14	7
média arit.	5.3	54.6	8.7	3

A partir dos valores apresentados nas tabelas, verifica-se que o *DTM* é muito efetivo para a correção de desvios preditos incorretamente [33], principalmente dos desvios incondicionais indiretos. Os valores percentuais correspondentes ao cache de instruções *L1* (*I-Cache*) e instruções de leitura à memória (*loads em D-cache*), representam o percentual de acessos (sobre o total de acessos efetuados) que deixaram de serem efetuados em decorrência do reuso explorado pelo *DTM*. Este valor é muito significativo e apresenta um indício de que o *DTM* pode ser um elemento efetivo na implementação de processadores com baixo consumo de energia [8, 31].

Na seqüência de avaliações a serem apresentadas, serão consideradas diferentes configurações do processador que incorpora o *DTM* e seus efeitos no reuso e ganho de performance. Para os experimentos, o efeito da correção de desvios pode ser desconsiderada quando utiliza-se um processador substrato configurado com um preditor de desvios perfeito, isolando assim, o desempenho do mecanismo *DTM* da performance do preditor de desvios. Será verificado também, o efeito produzido quando considera-se os caches perfeitos, e o conjunto caches perfeitos e predição de desvios perfeita.

Avaliação do *DTM* considerando um preditor de desvios perfeito.

As medidas a seguir, avaliam o impacto do mecanismo *DTM* no reuso e nos ganhos de performance obtidos quando o processador que o incorpora possui um preditor de desvios perfeito. Os experimentos realizados consideraram que todas as predições de desvios foram realizadas corretamente (direções e alvos). Os valores apresentados para a configuração *DTM* foram obtidos da subseção 5.2.3, enquanto os valores obtidos para a configuração $DTM_{pred.perf}$ foram obtidos considerando o processador substrato configurado com o preditor de desvios perfeito, e este mesmo processador configurado, incorporando o mecanismo *DTM*.

As Figuras 5.35 e 5.36 apresentam para o *SPECInt95* e *SPECFp95*, a variação de reuso considerando as configurações *DTM* e $DTM_{pred.perf}$ comparadas. Como o percentual de reuso é dado por $ir/itot$, os valores calculados diferem, pois $itot_{DTM} > itot_{DTMpred.perf}$, dado que $itot_{DTMpred.perf}$ é igual ao número de instruções entregues (consideração do preditor de desvios perfeito), enquanto $itot_{DTM}$ corresponde ao número de instruções entregues mais o número de instruções especulativas que foram descartadas. Esta ressalva, justifica parcialmente a pequena diferença favorável à configuração $DTM_{pred.perf}$. Entretanto, observações mais apuradas do comportamento dos programas considerando as duas configurações, identificaram um maior número de instruções redundantes na configuração *DTM*, quando esta é comparada à configuração $DTM_{pred.perf}$. Na realidade, ocorreu uma redução do número e instruções identificadas como redundantes e reusadas pela configuração $DTM_{pred.perf}$. Esta é justificada pelo efeito preditor de desvios perfeito, pois este afeta a disponibilidade dos operandos de entrada para instruções ou traces a serem reusados.

A Figura 5.34 esboça um exemplo que caracteriza o efeito negativo (ao reuso) do preditor de desvios perfeito. Considerando os círculos hachurados como sendo um trace que pode ser identificado como redundante e posteriormente reusado, observa-se que este trace depende somente do resultado produzido pela instrução de leitura à memória no endereço *100*. Para o caso em que a instrução *100* não produza um acerto ao acessar o cache de dados *L1*, o acesso ao cache *L2* provocará uma latência de 6 ciclos e o operando não estará disponível no momento da efetuação do teste de reuso para o trace iniciando em *114* (considerando ambas as instruções avaliadas em ciclos diferentes, pois um desvio tomado determina uma alteração no fluxo de busca). Para a consideração anterior, o trace não será reusado, pois o preditor de desvios indicou o caminho de execução correto para a instrução de desvio *108*, e as instruções a partir do endereço *114* foram avaliadas antecipadamente ao valor a ser produzido pela instrução *100*. Entretanto, na avaliação do mesmo caso e considerando um preditor de desvios real, uma incorreção na predição da instrução *108*, provocaria a execução de um caminho especulativamente incorreto a partir da instrução *10C*. Esta execução adicionada ao número de ciclos necessários para resolução do desvio (este também depende do valor produzido pela instrução *100*) e a penalidade decorrente do redirecionamento e descarte de instruções do pipeline, produzirão um atraso suficiente para que a instrução *100* produza um valor válido, e este esteja disponível quando do teste de reuso do trace avaliado. Embora muitas outras situações com conseqüências similares foram observadas, neste trabalho não será considerada uma avaliação exaustiva destas.

Avaliando os resultados expostos pelas Figuras 5.37 e 5.38, os ganhos de performance obtidos pela configuração $DTM_{pred.perf}$, foram reduzidos sensivelmente quando comparados com a configuração *DTM* com um preditor real. Esta redução já era esperada, visto que foram anulados os ganhos decorrentes da correção de desvios preditos incorretamente e as situações relacionadas a disponibilização de operandos de entrada comentadas no parágrafo anterior.

Para configuração $DTM_{pred.perf}$, os programas do *SPECInt95* apresentaram ganhos de performance variando de 3% a 13% para os programas *vortex* e *m88ksim* respectivamente e com média harmônica de 5%. Para a mesma configuração, os programas do *SPECFp95* apresentaram ganhos de performance variando de 1% a 8%

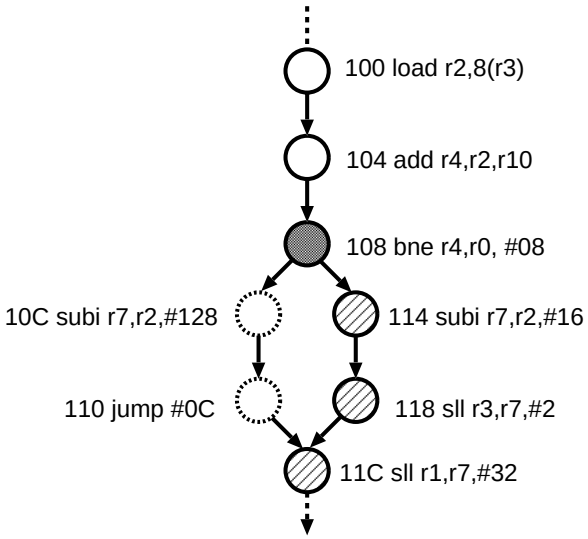


Figura 5.34: Exemplo do efeito do preditor de desvios perfeito no reuso de traces.

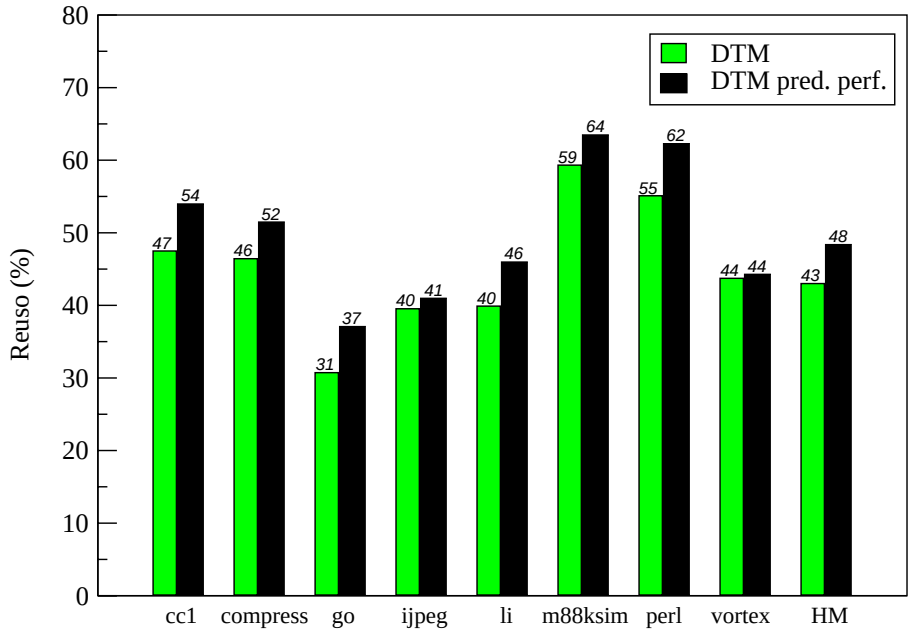


Figura 5.35: Variação no reuso explorado pelo *DTM* considerando um preditor de desvios perfeito, *SPECInt95*.

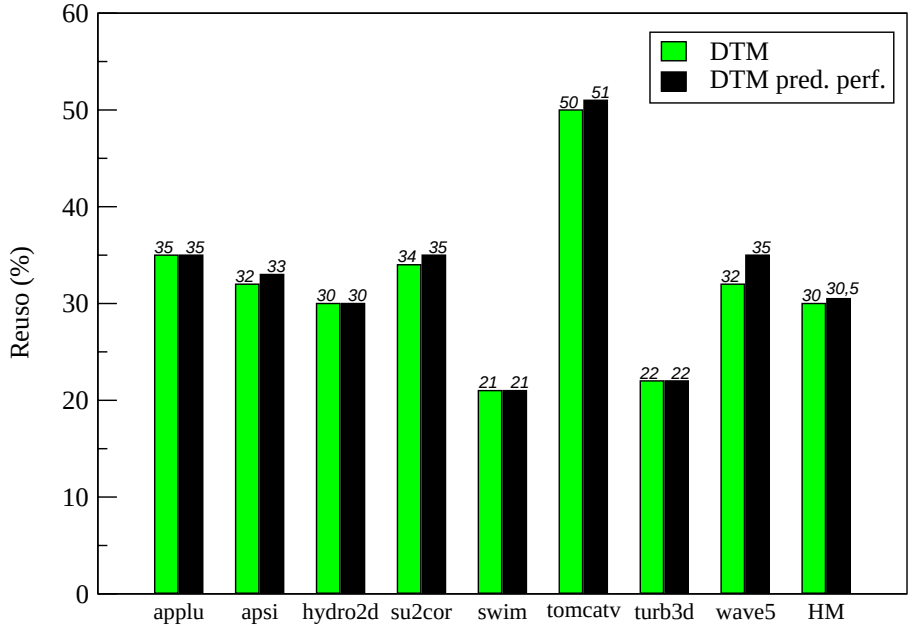


Figura 5.36: Variação no reuso explorado pelo *DTM* considerando um preditor de desvios perfeito, *SPECFp95*.

para os programas *wave5* e *tomcatv* respectivamente e com média harmônica de 2.7%. Comparando os resultados obtidos entre as configurações *DTM* e $DTM_{pred.perf}$, para a configuração $DTM_{pred.perf}$, todos os programas do *SPECInt95* e *SPECFp95* sofreram reduções acentuadas nos ganhos de performance obtidos. Os programas individualmente mais afetados foram *perl*, *li*, *wave5* e *swim*, estes apresentaram 66%, 50%, 91.5% e 71.5% respectivamente, de reduções percentuais de ganhos de performance quando comparados à configuração *DTM*. Na média (harmônica), os programas do *SPECInt95* e *SPECFp95*, apresentaram reduções percentuais de ganhos de performance de 40% e 61.5% respectivamente. Observações direcionadas aos programas *wave5* e *swim*, identificaram que estes apresentaram uma dramática redução dos ganhos de performance, visto que estes beneficiam-se extremamente da correção de desvios preditos incorretamente, particularmente dos desvios incondicionais indiretos que apresentam altos índices de incorreção na predição do endereço de destino (armazenado em um registrador).

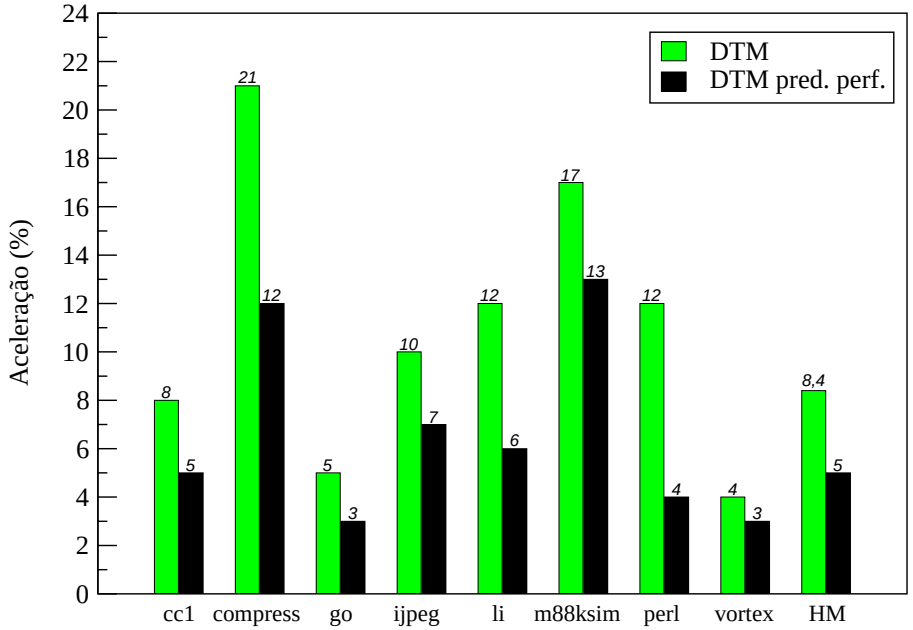


Figura 5.37: Variação nos ganhos de performance considerando um preditor de desvios perfeito, *SPECInt95*

Avaliação do *DTM* considerando caches perfeitos.

As medidas a seguir, avaliam o impacto do mecanismo *DTM* com relação aos caches de instruções e dados. Os experimentos realizados, consideram que todos os acessos feitos à hierarquia de memória foram satisfeitos pelos caches *L1* de instruções e dados, e que todos os acessos ocorrem com latência de 1 ciclo. A configuração com os caches perfeitos foi simulada para o processador substrato e para o processador substrato incorporando o mecanismo *DTM*, de modo a prover uma base comum para comparações.

As Figuras 5.39 e 5.40 apresentam para o *SPECInt95* e *SPECFp95*, a variação de reuso considerando a configuração de caches perfeitos. As comparações são efetuadas considerando o reuso obtido pelo *DTM* da subseção 5.2.3 e o $DTM_{cache-perf}$ configurado com caches perfeitos. Observa-se nos valores plotados por ambos os gráficos, que o reuso explorado pelo $DTM_{cache-perf}$ apresenta um pequeno decréscimo quando comparado ao reuso explorado pelo *DTM* sem caches perfeitos. Esta diferença decorre da variação de instruções e traces identificados como redundantes e reusados. Um trace que é reusado inúmeras vezes dado que foi habilitado por uma instrução de memória disponibilizando os valores de seu contexto de entrada, prova-

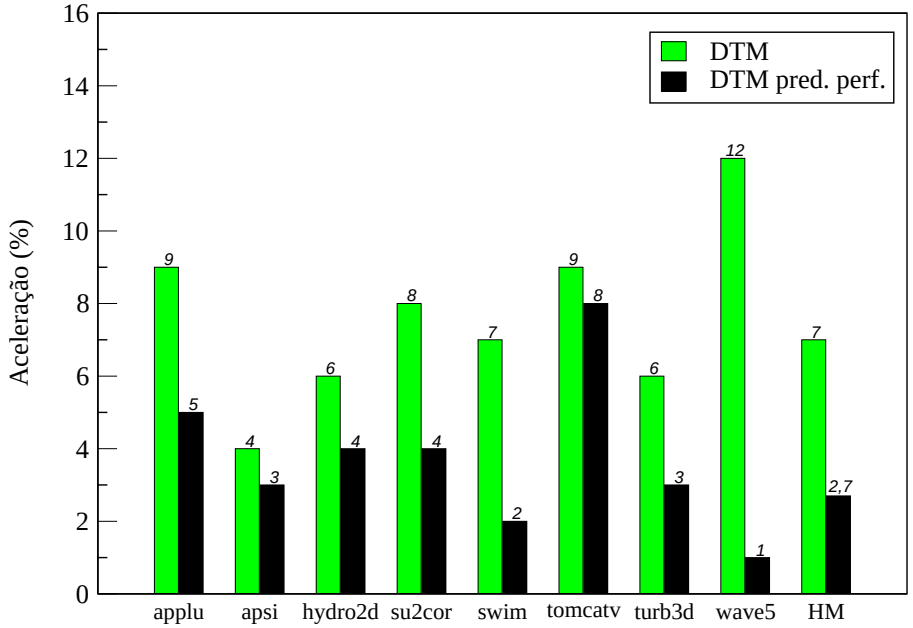


Figura 5.38: Variação nos ganhos de performance considerando um preditor de desvios perfeito, *SPECFp95*

velmente irá ocupar permanentemente (e com várias instâncias) a *Memo_Table_T*. Este trace poderá possuir poucas instruções (influenciando pouco na quantidade de reuso), entretanto ser bastante efetivo contribuindo significativamente nos ganhos de performance quando for reusado.

Os experimentos a seguir, consideram para a configuração $DTM_{cache-perf}$, o ganho de performance sobre o mesmo processador substrato com caches perfeitos.

Analisando o gráfico da Figura 5.41, observa-se altos ganhos de performance para os programas do *SPECInt95*. Este ganho de performance é proveniente de uma característica interessante, pois como já observado na subseção 5.3.1, uma grande quantidade de instruções de memória finalizam os traces construídos e estas instruções possuirão os seus endereços de acesso à memória antecipados quando do reuso dos traces que as incorporam. Supondo que estas instruções de memória produzam valores em um ciclo (acerto no cache de dados *L1*), o ganho proveniente do reuso do trace será efetivo, caso contrário (falha no acesso ao cache de dados *L1*), o ganho proveniente do reuso do trace pode não ser completamente efetivo, pois a penalidade imposta pelo acesso aos outros elementos da hierarquia de memória (latências inseridas), podem anular os ganhos decorrentes do reuso do trace. Foram

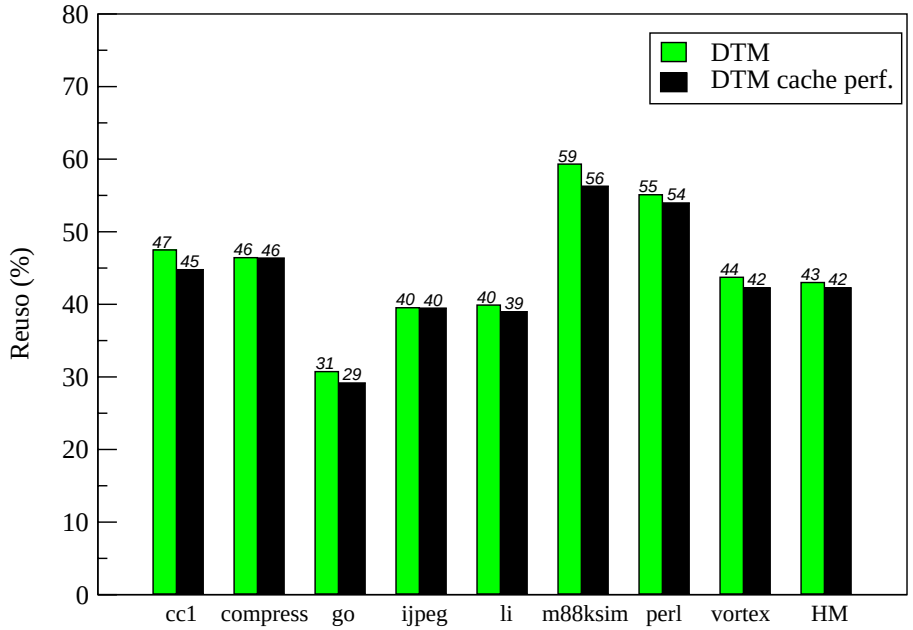


Figura 5.39: Variação no reuso considerando caches perfeitos, *SPECInt95*.

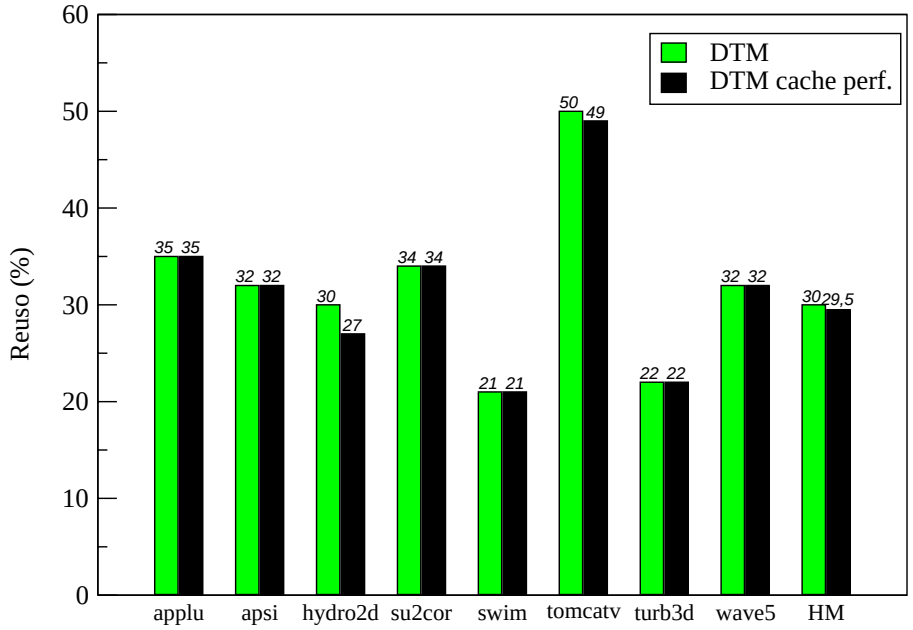


Figura 5.40: Variação no reuso considerando caches perfeitos, *SPECFp95*.

observados ganhos de performance para todos os programas avaliados e com média harmônica de 11%. Estes ganhos foram mais pronunciados para os programas *cc1* com 12% e *perl* com 19%. É importante ressaltar, que mesmo para programas apresentando taxas de falha ao acesso ao cache *L1* muito baixas, os traces podem se concentrar em tornos destas falhas.

Analisando o gráfico da Figura 5.42, observa-se a ocorrência de ganhos de performance para a maioria dos programas do *SPECFp95*. Considerações já mencionadas para a análise dos programas do *SPECInt95*, são válidas também para o *SPECFp95*. Entretanto, dois programas apresentaram o desempenho reduzido, estes foram o *hydro2d* e o *su2cor* com aceleração de 1% e 7% respectivamente. Uma análise mais apurada da execução do programa *hydro2d*, revelou que acessos sem sucesso aos caches de instruções, promovem atrasos que favorecem a disponibilidade de operandos prontos para os traces e instruções redundantes (estes atrasos não ocorrem na configuração de caches perfeitos), e que este programa possui seus ganhos de performance calcados na antecipação de instruções representadas nos traces e que ainda não foram trazidas do cache de instruções (em decorrência de acessos sem sucesso). Para o programa *su2cor*, ocorreu um equilíbrio entre os efeitos positivos e negativos dos acessos efetuados com sucesso aos caches, sendo que a redução no ganho de performance não é significativa quando considerando a configuração com caches perfeitos (1%). Finalmente, observa-se que a média harmônica de 4.2% relativa ao ganho de performance total foi afetada exclusivamente pelo resultado do programa *hydro2d*.

Avaliação do *DTM* considerando caches e preditor de desvios perfeitos.

As medidas a seguir, consideram a conjunção das configurações anteriores, ou seja, o processador substrato será acrescido de caches perfeitos e de um preditor de desvios perfeito. Sobre este, será incorporado o mecanismo *DTM* e esta configuração será denominada DTM_{ideal}. O objetivo deste experimento é verificar o desempenho do *DTM*, isolando-o dos efeitos impostos pelos caches e preditor de desvios.

As Figuras 5.43 e 5.44 apresentam os valores de reuso obtidos para a configuração DTM_{ideal}. Estes, são comparados aos valores obtidos pelo *DTM* da subseção 5.2.3. Observa-se nos gráficos, que os valores de reuso para os programas do *SPECInt95*,

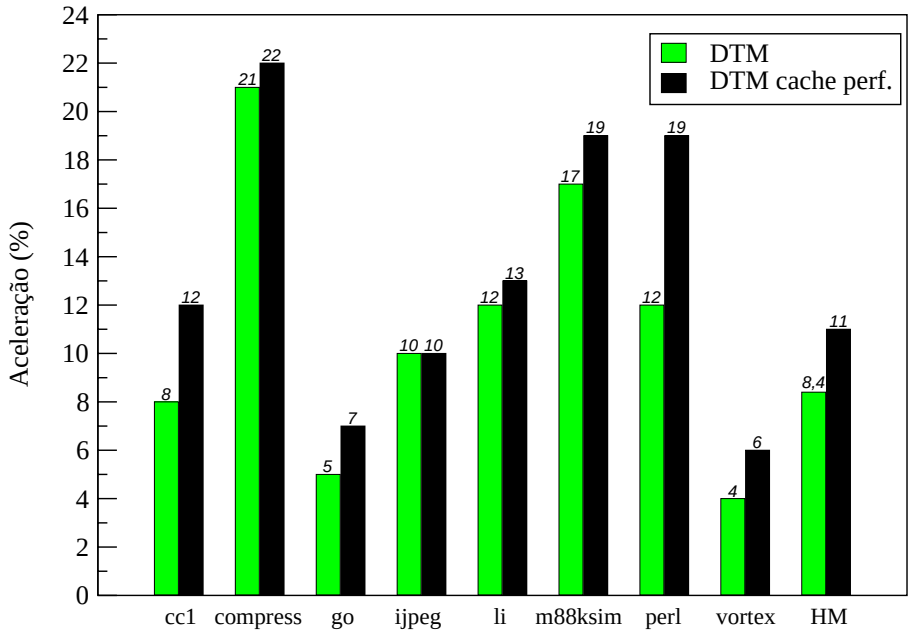


Figura 5.41: Variação nos ganhos de performance considerando caches perfeitos, *SPECInt95*.

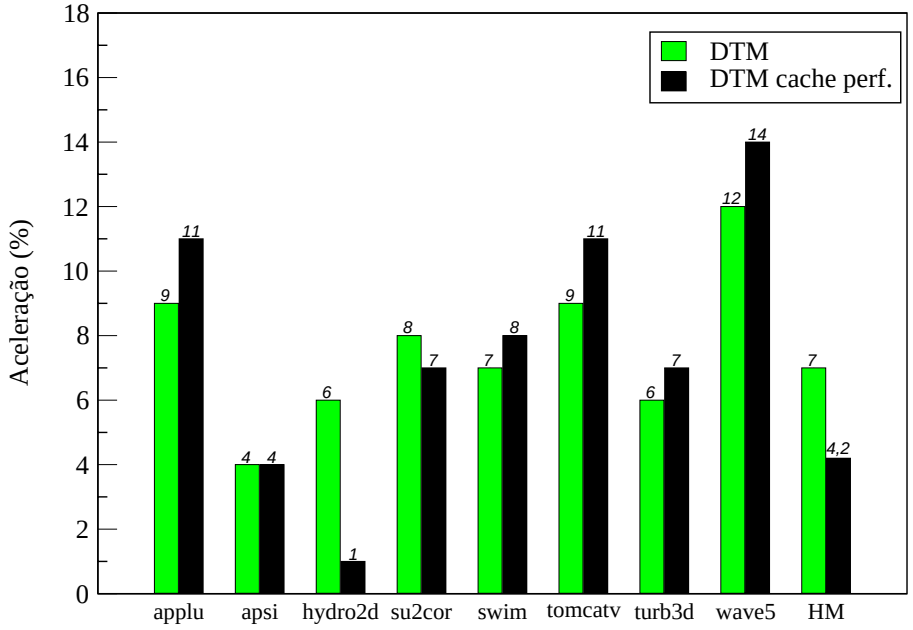


Figura 5.42: Variação nos ganhos de performance considerando caches perfeitos, *SPECFp95*.

variaram de 36% a 62% para os programas *go* e *m88ksim*, e com média harmônica de 47.4%. Enquanto para os programas do *SPECFp95* os valores variaram de 21% a 50% para os programas *swim* e *tomcatv*, e com média harmônica de 30.2%.

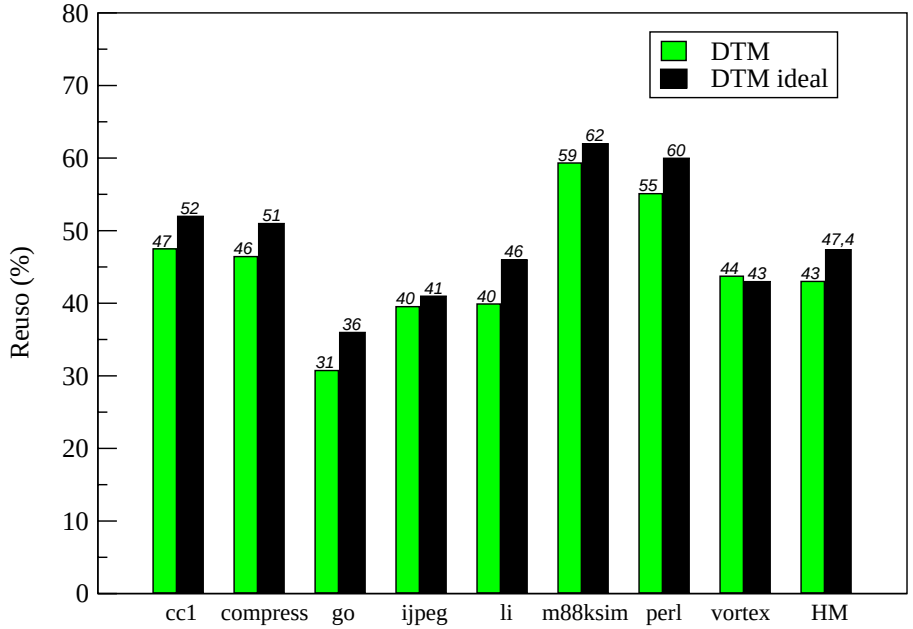


Figura 5.43: Variações no reuso considerando caches e preditor de desvios perfeitos, *SPECInt95*.

As Figuras 5.45 e 5.46, apresentam os valores de ganho de performance obtidos para a configuração DTM_{ideal}. Observa-se que os valores de ganho de performance para o *SPECInt95*, variaram de 5% a 21% para os programas *go* e *compress*, e com média de 7.6%. Enquanto para o *SPECFp95* os valores variaram de 1% a 9% para os programas *wave5* e *tomcatv*, e com média de 2.7%.

As avaliações efetuadas considerando o DTM_{ideal}, identificaram valores de reuso semelhantes aos valores obtidos pelas configurações $DTM_{pred.perf}$ e $DTM_{cache-perf}$. Desde que os valores destas configurações são muito semelhantes, os resultados obtidos não apresentaram nenhuma alteração sensível. Entretanto, como esperado, os ganhos de performance para configuração DTM_{ideal} sofreram alterações, visto que existem diferenças entre os valores apresentados para as configurações isoladas $DTM_{pred.perf}$ e $DTM_{cache-perf}$. Para os programas do *SPECInt95*, a média harmônica de 7.6% para a configuração DTM_{ideal}, está situada entre os valores médios de 5% e 11% obtidos nas configurações $DTM_{pred.perf}$ e $DTM_{cache-perf}$ res-

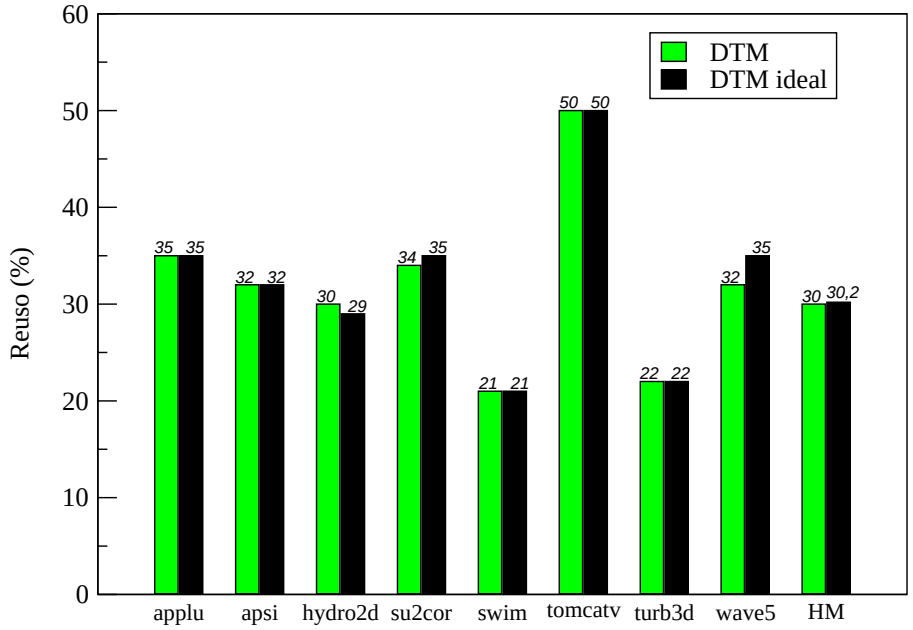


Figura 5.44: Variações no reuso considerando caches e preditor de desvios perfeitos, *SPECFp95*.

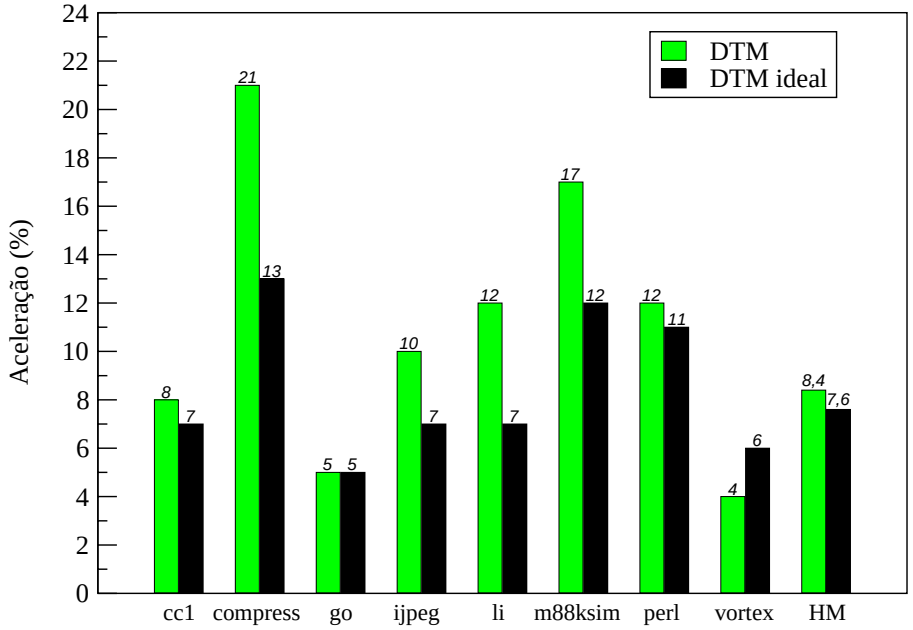


Figura 5.45: Variações nos ganhos de performance considerando caches e preditor de desvios, *SPECInt95*.

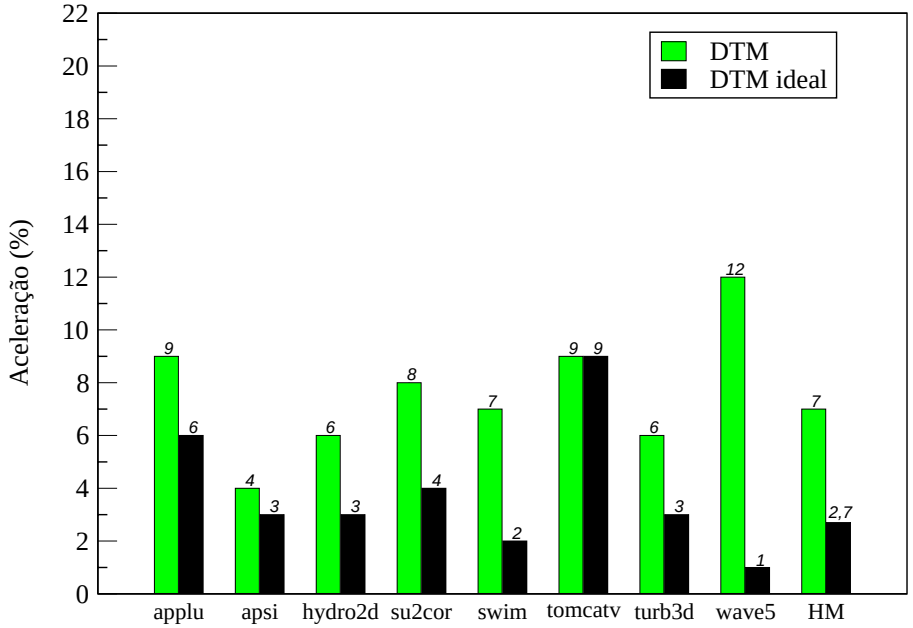


Figura 5.46: Variações nos ganhos de performance considerando caches e preditor de desvios perfeitos, *SPECFp95*.

pectivamente. Este resultado, descreve a ocorrência de um equilíbrio entre os efeitos dos caches perfeitos e do preditor de desvio perfeito. Entretanto, esta mesma consideração não pode ser estabelecida para os programas do *SPECFp95*, dado que estes programas em média, beneficiam-se muito de incorreções nas predições de desvios, e não se beneficiam (uniformemente) da configuração de caches perfeitos.

5.3.8 Reusando todos os traces redundantes em *Memo_Table_T*.

Nesta subseção, o mecanismo *DTM* será avaliado quanto ao seu potencial de reuso e ganho de performance, considerando o caso em que todos os traces redundantes e presentes em *Memo_Table_T* são reusados. Esta avaliação é motivada pela observação de que muitos traces deixaram de serem reusados em decorrência da indisponibilidade de seus operandos do contexto de entrada (no processador) quando da efetuação do teste de reuso (este é efetuado apenas uma vez).

Como já descrito anteriormente, um trace (presente *Memo_Table_T*) é reusado apenas quando ambas as condições abaixo ocorrem:

(i) Os valores dos operandos de seu contexto de entrada estão disponíveis no processador;

(ii) Os valores de (i) são identificados como idênticos ao contexto do processador.

É imediato observar que a condição (ii) não sendo satisfeita, determina completamente a impossibilidade de reuso de um trace. Entretanto, a condição (i) não sendo satisfeita, não impede a possibilidade de existência de um trace em *Memo_Table_T* que poderia ser identificado como redundante e posteriormente reusado. O seguinte exemplo, explicita detalhadamente um caso relacionado à condição (i). Supondo um trace t que possua n instâncias armazenadas em *Memo_Table_T* e que seu contexto de entrada seja composto pelos operandos op_1 e op_2. Supondo ainda que durante o ciclo i seja efetuado o teste de reuso, e que neste ciclo o valor correspondente ao operando op_1 esteja disponível, enquanto o valor do operando op_2 não (estará disponível em um ciclo $i + k$). Para este caso, o trace t não será reusado de acordo com a condição (i), mesmo existindo em *Memo_Table_T* o trace t instanciado com os mesmos valores atribuídos a op_1 (no ciclo i) e op_2 (no ciclo $i + k$).

Para os experimentos a serem expostos e avaliados, foi considerada a existência de um *oráculo*. O *oráculo* proposto possui como função, resolver os impedimentos ao reuso causados pela condição (i). Este será ativado quando a condição (i) for identificada, e escolherá dentre todas as instâncias do trace que estão armazenadas em *Memo_Table_T*, aquela que possua o contexto de entrada que será idêntico ao contexto do processador quando este possuir os correspondentes operandos disponíveis, permitindo deste modo o reuso do trace. É importante ressaltar, que o oráculo aqui considerado não irá interferir na construção de traces, nem tampouco na *Memo_Table_G*, e que somente irá determinar o trace a ser reusado se o mesmo estiver presente em *Memo_Table_T*.

Nos experimentos, foram considerados duas configurações para prover uma comparação. A primeira configuração é composta por um processador incorporando o mecanismo *DTM* descrito na subseção 5.2.3, a segunda configuração inclui o mesmo mecanismo *DTM* acrescido do oráculo, sendo esta última denominada DTM_{pvp}.

As Figuras 5.47 e 5.48 apresentam comparativamente, o reuso obtido considerando a configuração *DTM* e a configuração *DTMpvp*. Para o *SPECInt95* ocorreu um aumento significativo (para todos os programas) do percentual de reuso explorado pela configuração *DTMpvp*. Os valores obtidos variaram de 43% a 83% para

os programas *go* e *m88ksim* respectivamente. Na média, a configuração *DTMpvp* apresentou um percentual de reuso de 59% contra os 43% da configuração *DTM*. A mesma análise para o *SPECFp95*, identificou um pequeno aumento no percentual de reuso explorado. Os valores obtidos variaram de 23% a 56% para os programas *swim* e *tomcatv* respectivamente, enquanto a média variou de 30% para 34% considerando a configuração *DTM* e *DTMpvp* respectivamente. A partir dos valores apresentados, verifica-se que o reuso explorado nos programas do *SPECInt95* são mais sensíveis as restrições impostas pela condição (i), enquanto os programas do *SPECFp95* não são afetados na mesma proporção.

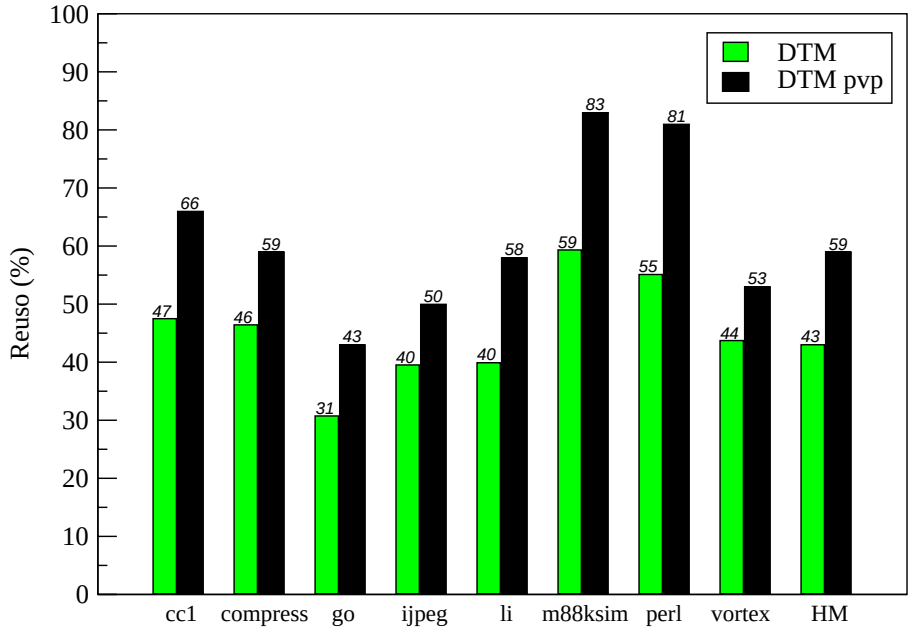


Figura 5.47: Reuso explorado pela configuração *DTMpvp*, *SPECInt95*.

As Figuras 5.49 e 5.50 apresentam os ganhos de performance obtidos pela configuração *DTMpvp*. Expressivos ganhos de performance foram obtidos para a totalidade dos programas do *SPECInt95*. Os valores variaram de 8% a 44% para os programas *vortex* e *m88ksim* respectivamente. Em média, a configuração *DTMpvp* apresentou um ganho de performance de 19.3% contra os 8.4% da configuração *DTM*. Para os programas do *SPECFp95*, também ocorreu um aumento do ganho de performance para todos os programas avaliados. Os valores variaram de 6% a 19% para os programas *apsi* e *wave5* respectivamente. Em média, a configuração *DTMpvp* apresentou um ganho de performance de 10% contra os 7% da configuração *DTM*.

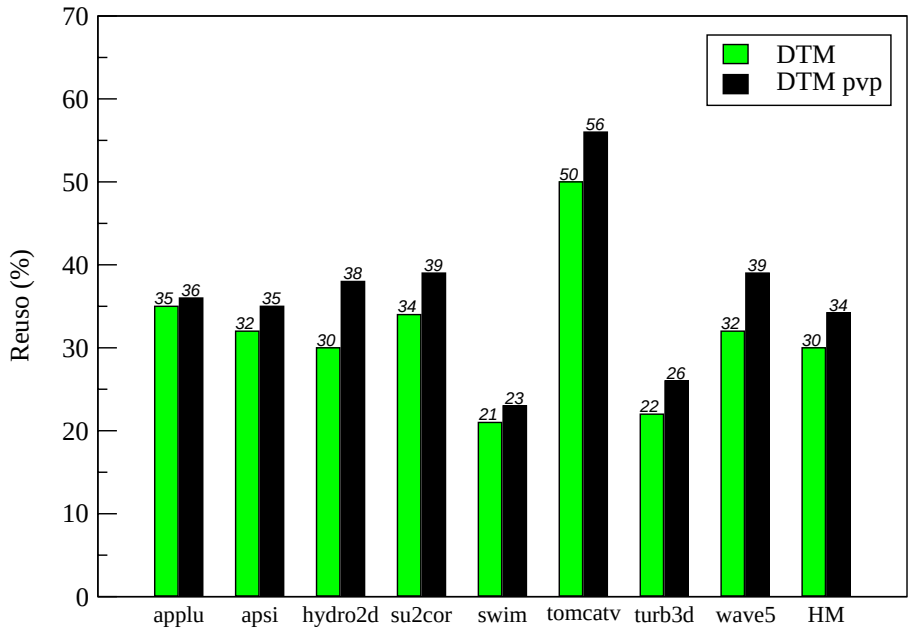


Figura 5.48: Reuso explorado pela configuração *DTMpvp*, *SPECFp95*.

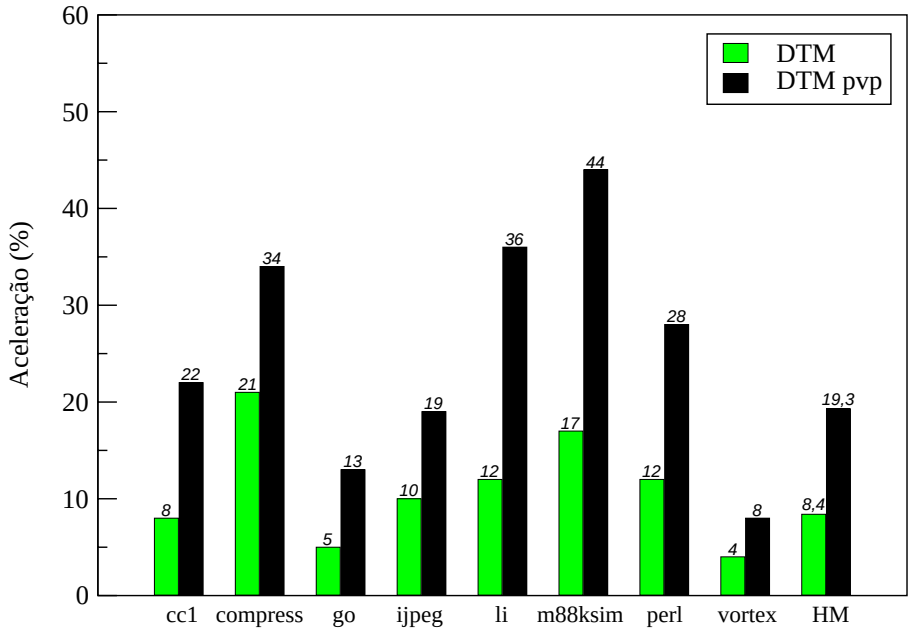


Figura 5.49: Ganho de performance obtido pela config. *DTMpvp*, *SPECInt95*.

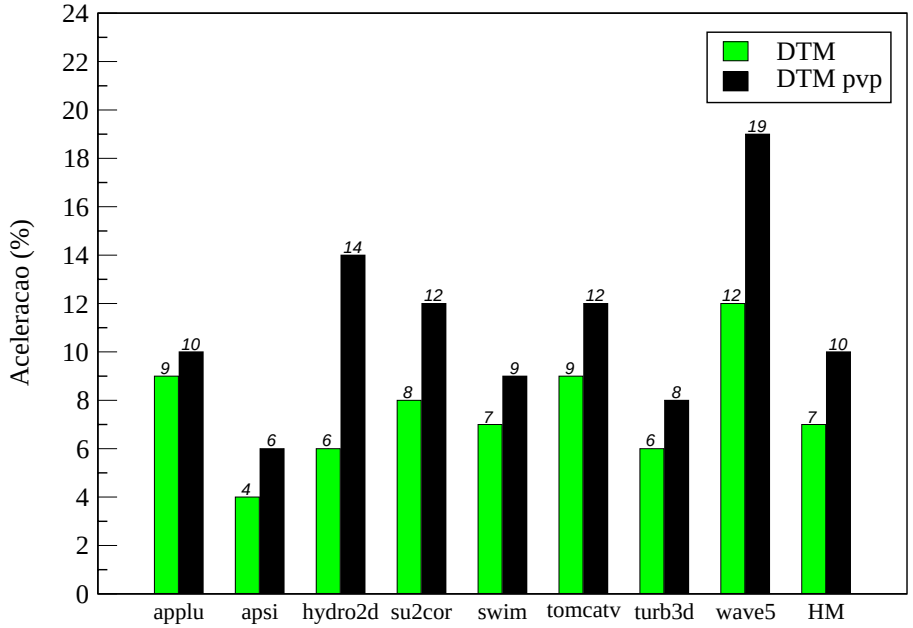


Figura 5.50: Ganho de performance obtido pela config. *DTMpvp*, *SPECFp95*.

Os valores de reuso e ganho de performance obtidos pelos experimentos realizados, revelaram que um pequeno percentual de traces que não foram reusados pela indisponibilidade de seus operandos do contexto de entrada, proporcionam significativos ganhos de performance quando reusados. Para o *SPECInt95*, um aumento médio de 37% do percentual de reuso identificado (de 43% para 59%), foram responsáveis por um aumento médio de aproximadamente 129% no ganho de performance (de 8.4% para 19.3%). Enquanto para o *SPECFp95*, um aumento médio de 13% do percentual de reuso identificado (de 30% para 34%), foram responsáveis por um aumento médio de aproximadamente 43% no ganho de performance (de 7% para 10%).

Os traces reusados que foram determinados pelo oráculo, apresentaram a restrição imposta pela condição (i). São exemplos de situações onde se verifica a ocorrência desta condição:

• Instruções de acesso à memória impedidas de produzirem os valores para os operandos do contexto de entrada de um trace, em decorrência de não possuírem suas requisições de acesso ao cache *L1* satisfeitas, ou não obtiveram recursos necessários para sua finalização (porta de leitura disponível);

• Instruções de desvio, que dependendo do resultado de sua predição, determinam

em que ciclos outras instruções dependentes do fluxo de execução serão executadas e produzirão resultados que disponibilizam valores de entrada para os traces;

• Quando instruções redundantes utilizadas para a construção do trace são instanciadas e identificadas como redundantes em ciclos diferentes, ou seja, os valores que as instanciaram são produzidos em ciclos diferentes.

O procedimento efetuado pelo *oráculo* responsável pela escolha do trace, é equivalente a um preditor de valores perfeito, isto é, um preditor que efetua predições sempre corretas para os valores a serem instanciados aos traces restritos pela condição (i). Implementações realísticas, poderiam estender o conceito de predição de valores [44, 29] para traces ao invés de instruções, e aplicá-la conjugadamente ao reuso. Para esta consideração, quando da ocorrência de uma predição de valores aplicada a traces, pode-se reusar os traces preditos de forma especulativa e verificar a correção da predição quando ocorrer a instanciação exata dos valores preditos. Para predições corretas, o contexto de saída despachado será marcado como não especulativo e entregue normalmente. Para predições incorretas, o contexto de saída despachado será descartado, invalidando assim os efeitos da predição de valores aplicada a traces redundantes.

Apesar de permitir o reuso de traces que não possuam seus operandos do contexto de entrada prontos, este tipo de predição também insere penalidades que podem afetar negativamente o desempenho, caso as predições efetuadas forem incorretas. Entretanto, a exatidão de predições de valores aplicada a traces (no contexto *DTM*), revelam-se em um cenário diferente da predição de valores usual. Diferente da predição de valores aplicado a instruções:

• Os traces a serem preditos (inseridos em *Memo_Table_T*) foram construídos por um critério calcado em redundâncias, ou seja, existe um critério mais apurado para suportar com uma maior exatidão a efetuação das predições;

• A predição de valores aplicada a traces pode ser empregada somente em situações especiais e restrita a traces que são afetados pela condição (i);

• A predição de valores aplicada a traces no contexto *DTM*, atua sobre uma tabela com poucas entradas (*Memo_Table_T* com 512 entradas);

• A seleção de traces candidatos a predição pode ser restrita a traces que possuam pelo menos uma quantidade fixa de operandos do contexto de entrada satisfazendo

a condição (ii), ou seja, a quantidade de possíveis traces candidatos é reduzida sensivelmente.

5.3.9 Aplicando o mecanismo *DTM* a processadores configurados com diferentes larguras

Nesta subseção será avaliado o impacto do *DTM* quando incorporado em processadores com diferentes larguras. Esta avaliação possui como objetivo, investigar a viabilidade de outras alternativas de configuração.

Serão avaliadas comparativamente, três configurações do mesmo processador substrato e suas respectivas versões incorporando o mecanismo *DTM*. A tabela 5.10 apresenta as diferentes configurações e os parâmetros escolhidos. Os parâmetros alterados correspondem a quantidade de instruções que são tratadas em cada estágio do pipeline. Para prover uma comparação mais equilibrada, as configurações com larguras *w=1* e *w=2* foram dotadas da capacidade de suportar a entrega de 4 instruções por ciclo. Este relaxamento é assumido para prover um melhor aproveitamento dos traces redundantes para todas as configurações. Para a configuração *w=2*, o *buffer de emissão/reordenação* e a *fila de load/store* possuem o mesmo número de entradas adotado pela configuração *w=4*. Enquanto para configuração *w=1*, este buffer foi reduzido para se aproximar de uma configuração escalar real. Os outros elementos da arquitetura, como caches, preditor de desvios, etc..., foram mantidos inalterados para todas as configurações.

Tabela 5.10: Configurações do processador para as diferentes larguras.

LARGURAS	**w=1**	**w=2**	**w=4**
Estágio de busca	1	2	4
Estágio de decodificação	1	2	4
Estágio de entrega	4	4	4
Buffer de emissão/reordenação	4	16	16
Fila de load/store	2	8	8

Nas Figuras 5.51 e 5.52, são apresentados os percentuais de reuso obtidos para cada configuração. Analisando os valores plotados para os programas do *SPECInt95*, nestes foram obtidos em média, valores de reuso de 60%, 48% e 43%, para as confi-

gurações *w=1*, *w=2* e *w=4* respectivamente. Observa-se que do percentual de reuso explorado é decrescente na medida em que as larguras são crescentes. Uma constatação interessante recai sobre a configuração *w=1*. Esta, apresenta praticamente o mesmo percentual de reuso explorado quando da inclusão do *oráculo* efetuada em 5.3.8. Esta característica reafirma a restrição imposta pela indisponibilidade dos operandos do contexto de entrada dos traces redundantes, e que ocorre principalmente na configuração *w=4*. Estas restrições são praticamente anuladas quando a configuração *w=1* é utilizada. Análise análoga para os programas do *SPECFp95*, identificaram em média, valores de reuso de 37%, 32% e 32%, para as configurações *w=1*, *w=2* e *w=4* respectivamente. As mesmas considerações já expostas para o *SPECInt95* são válidas para o *SPECFp95*, com exceção das configurações *w=2* e *w=4*, que apresentaram o mesmo percentual de reuso.

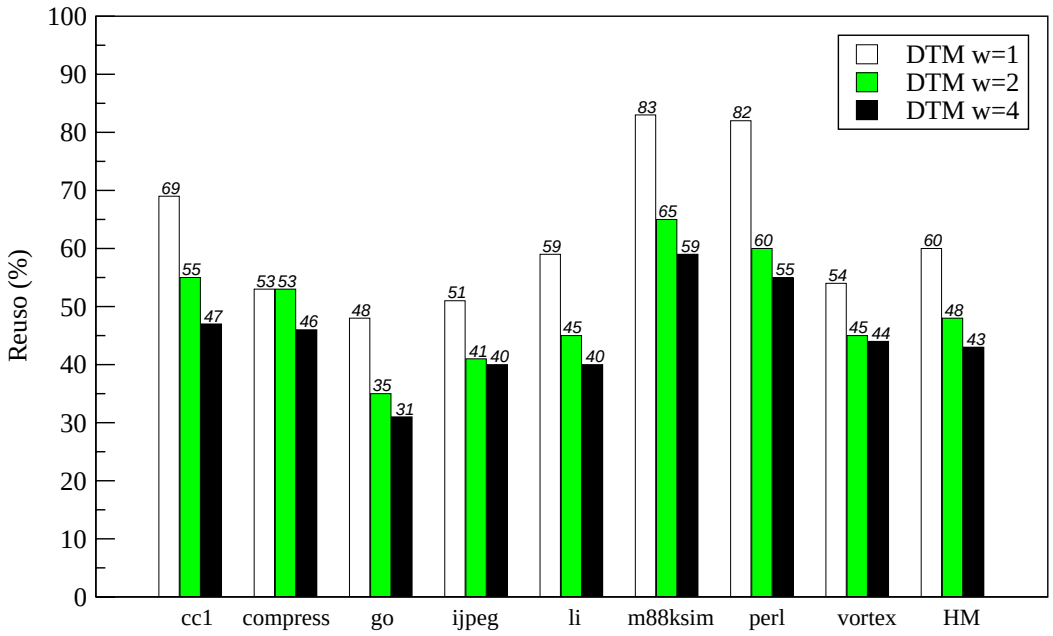


Figura 5.51: Reuso explorado pelo *DTM* em processadores com diferentes larguras, *SPECInt95*.

Considerando os resultados expostos pela Figura 5.53, para os programas do *SPECInt95* foram obtidos em média, ganhos de performance de 25%, 19% e 8.4% para as configurações *w=1*, *w=2* e *w=4* respectivamente. Para os programas *cc1* e *ijpeg*, a configuração *w=2* apresentou ganhos de performance maiores que as demais (19% para *cc1* e 25% para *ijpeg*). Para os outros programas, a configuração *w=1*

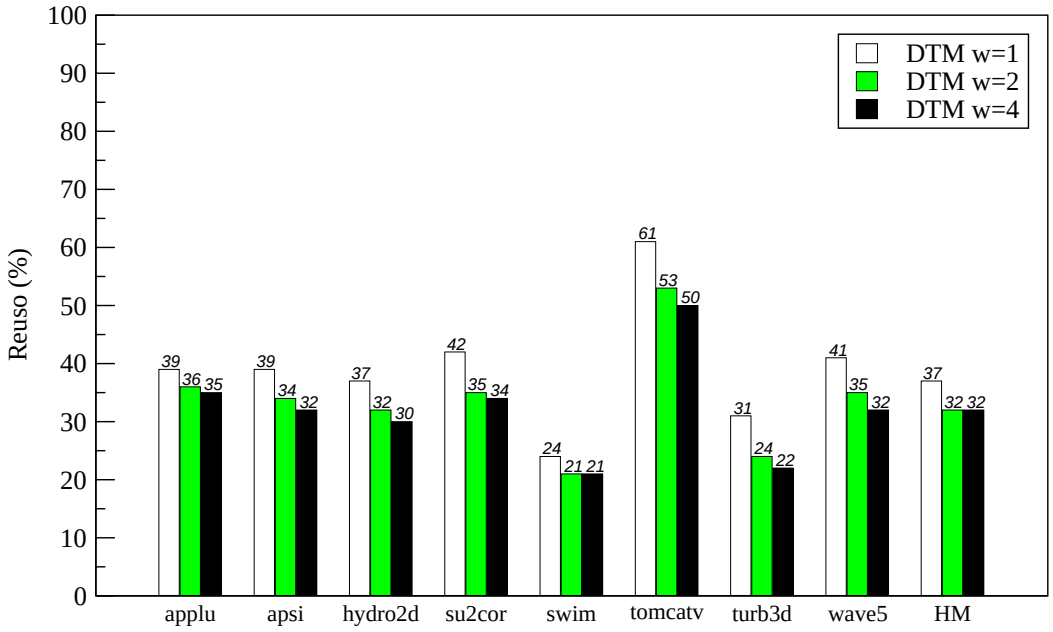


Figura 5.52: Reuso explorado pelo *DTM* em processadores com diferentes larguras, *SPECFp95*.

apresentou os maiores ganhos, seguida da configuração *w=2* e *w=4* (nesta ordem). Os resultados obtidos eram esperados, visto que as configurações com larguras maiores, agravam mais acentuadamente a disponibilidade dos valores dos operandos do contexto de entrada dos traces redundantes. Esta característica gerou investigações que constataram um significativo aumento no número de traces que foram reusados quando as larguras são reduzidas. Os efeitos produzidos nos ganhos de performance pelo reuso de traces, são distintos para as diferentes configurações avaliadas. Por exemplo, reusando traces com 3 instruções que atribuem valores imediatos aos registradores de destino (contexto de entrada 0), são extremamente valiosos para a configuração *w=1*, pois evitam 3 (três) execuções escalares, enquanto para a configuração *w=2* são evitadas 2 (duas) execuções e finalmente para a configuração *w=4* é evitada apenas 1 (uma) execução (desconsiderando para todos os casos, as limitação de unidades funcionais disponíveis).

Considerando os resultados expostos pela Figura 5.54, para os programas do *SPECFp95* foram obtidos em média, ganhos de performance de 9%, 13% e 7% para as configurações *w=1*, *w=2* e *w=4* respectivamente. A configuração *w=2* apresentou um maior ganho de performance, seguida da configuração *w=1* e da configuração

w=4. Uma excessão ocorreu para o programa *swim*, neste, o ganho de performance da configuração *w=4* foi maior que o da configuração *w=1*. Os resultados obtidos diferem do comportamento observado para os programas do *SPECInt95*. Contrariamente a este, o desempenho da configuração *w=2* foi maior que o desempenho da configuração *w=1*. Este comportamento somente poderá ser justificado avaliando-se as situações em que traces foram reusados e a composição dos mesmos para as diferentes configurações apresentadas.

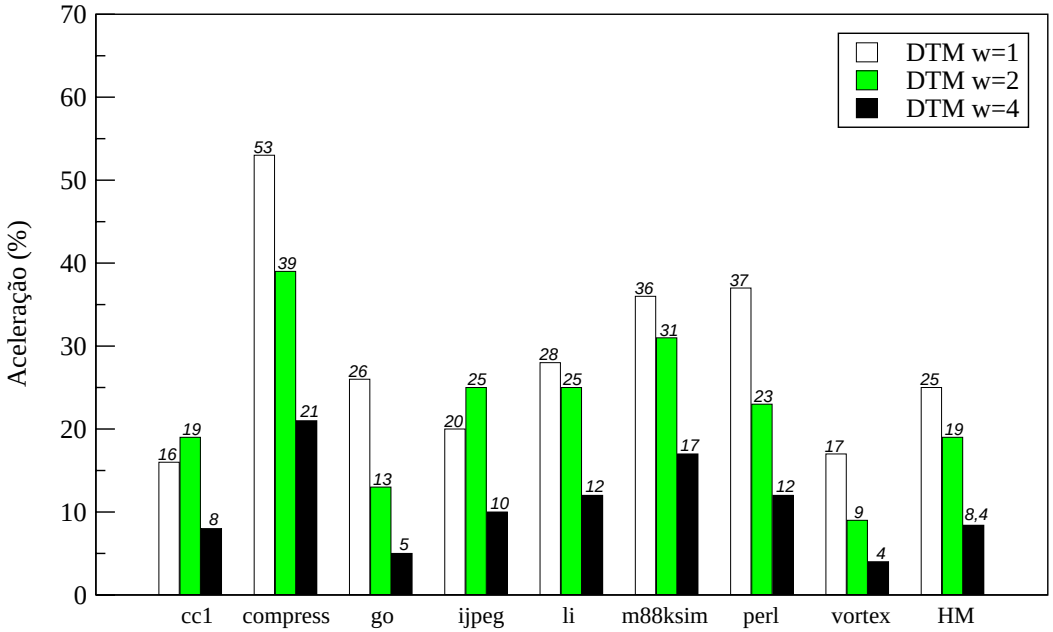


Figura 5.53: Ganhos de performance obtidos pelo *DTM* em processadores com diferentes larguras, *SPECInt95*.

As medidas a seguir, plotam os valores de *ipc* obtidos para as configurações *w=1 base*, *w=2 base* e *w=4 base*, considerando o processador substrato como base e, *w=1 DTM*, *w=2 DTM* e *w=4 DTM* considerando o mesmo processador base incorporando o *DTM*. O objetivo desta exposição é identificar o quanto distam em termos de *ipc* as configurações avaliadas, obtendo desta forma informações comparativas de modo global.

A Figura 5.55 apresenta os valores de *ipc* plotados para cada um dos programas do *SPECInt95*. Observa-se que os valores plotados para a configuração *w=2 DTM*, aproximam-se acentuadamente dos valores obtidos pela configuração *w=4 base*. A média harmônica obtida entre os valores de *ipc* para cada configuração revelaram que

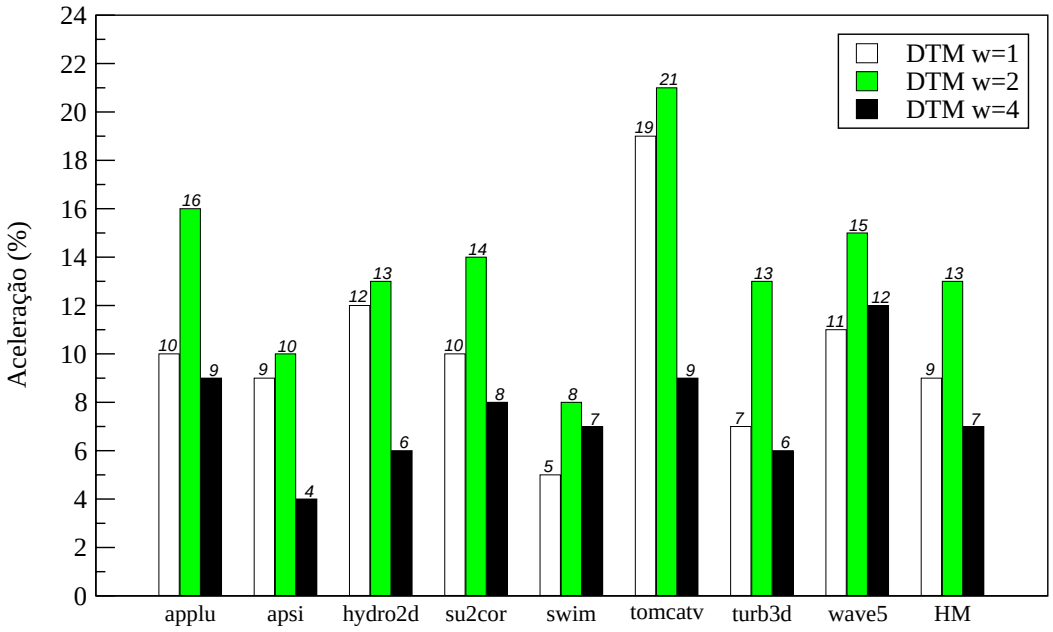


Figura 5.54: Ganhos de performance obtidos pelo *DTM* em processadores com diferentes larguras, *SPECFp95*.

comparativamente: a configuração *w=4 base* apresentou um ganho de performance de apenas 8.6% sobre a configuração *w=2 DTM*; a configuração *w=2 base* apresentou um ganho de performance de 27% sobre a configuração *w=1 DTM* e a configuração *w=4 base* apresentou um ganho de performance de 66% sobre a configuração *w=1 DTM*.

A Figura 5.56 apresenta os valores de *ipc* plotados para cada um dos programas do *SPECFp95*. Para o *SPECFp95*, os valores plotados para a configuração *w=2 DTM* aproximam-se ainda mais acentuadamente dos valores obtidos pela configuração *w=4 base*. Aplicando a mesma análise feita anteriormente para o *SPECInt95*, a média harmônica obtida entre os valores de *ipc* para cada configuração revelaram que comparativamente: a configuração *w=4 base* apresentou um ganho de performance de apenas 2.5% sobre a configuração *w=2 DTM*; a configuração *w=2 base* apresentou um ganho de performance de 87% sobre a configuração *w=1 DTM* e a configuração *w=4 base* apresentou um ganho de performance de 92% sobre a configuração *w=1 DTM*.

Pôde-se observar claramente que a configuração *w=2 DTM* consegue obter de forma equilibrada, valores de reuso e ganhos de performance bastante atraentes,

principalmente com relação aos programas do *SPECFp95*. Os valores obtidos para as configurações avaliadas fornecem uma base comparativa, que podem determinar alternativas de projeto (dependendo da aplicação) para processadores, visando reduções de complexidade, custo e consumo de potência [51, 3, 38].

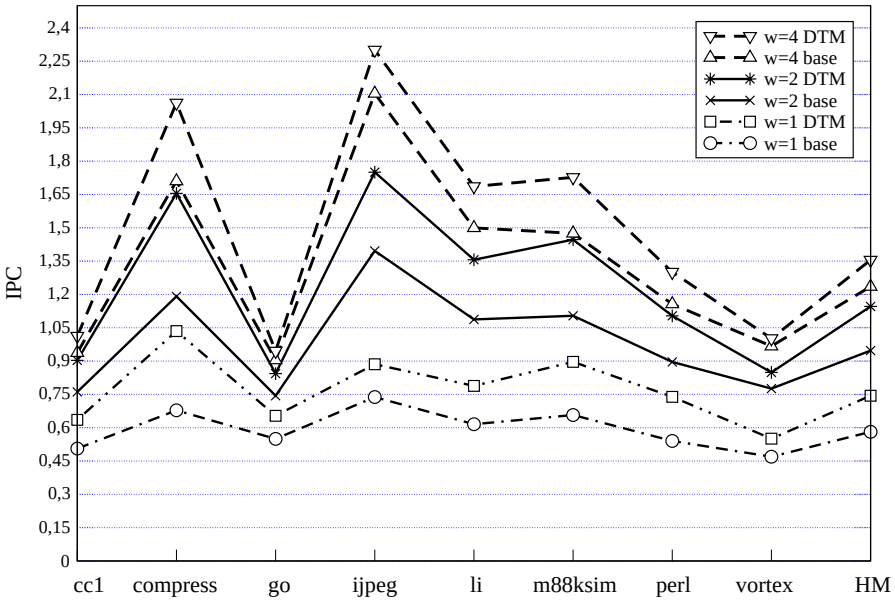


Figura 5.55: Valores de *ipc* para processadores incorporando o *DTM* e com diferentes larguras, *SPECInt95*.

5.3.10 Avaliação do *DTM* considerando um processador substrato configurado com parâmetros arquiteturais mais agressivos

O mecanismo *DTM* foi avaliado durante todo este trabalho, considerando o processador substrato configurado de forma conservadora. Os experimentos a seguir, irão avaliar o *DTM* incorporado-o a um processador substrato com alterações agressivas em seus parâmetros arquiteturais. O processador a ser simulado possui alterações nos seguintes parâmetros:

- Os caches de instruções e dados *L1* foram configurados com 128 kbytes (para cada um);
- O preditor de desvios foi configurado com 16K entradas;

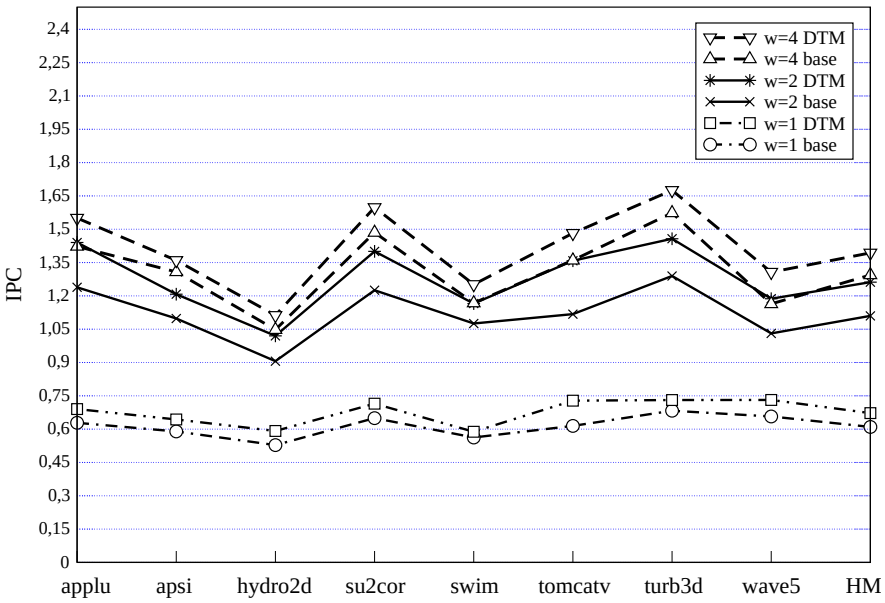


Figura 5.56: Valores de *ipc* para processadores incorporando o *DTM* e com diferentes larguras, *SPECFp95*.

• Inclusão de mais 2 unidades de Load/Store, proporcionando basicamente, o aumento de portas de leitura e escrita no cache de dados L1 (serão agora quatro portas compartilhadas para as funções de leitura/escrita).

O mecanismo *DTM* incorporado a esta nova configuração, será denominado *DTM+*

As Figuras 5.57 e 5.58 apresentam para o *SPECInt95* e *SPECFp95* respectivamente, a variação de reuso observada para a configuração *DTM+*. Comparativamente ao mecanismo *DTM* com a configuração conservadora (subseção 5.2.3), a configuração *DTM+* apresentou um pequeno decréscimo no percentual de reuso explorado. Para o *SPECInt95*, o percentual de reuso explorado variou de 29% a 56% para os programas *go* e *m88ksim* respectivamente, e apresentou média de 42%. Enquanto para o *SPECFp95*, o percentual de reuso explorado variou de 23% a 49% para os programas *swim* e *tomcatv* respectivamente, e apresentou média de 29.6%.

As Figuras 5.59 e 5.60 apresentam para o *SPECInt95* e *SPECFp95* respectivamente, a variação de ganhos de performance observados para a configuração *DTM+*. Comparativamente ao mecanismo *DTM* com a configuração conservadora (subseção 5.2.3), a configuração *DTM+* apresentou ganhos de performance notadamente superiores considerando o *SPECInt95*. Este, apresentou ganhos de performance variando

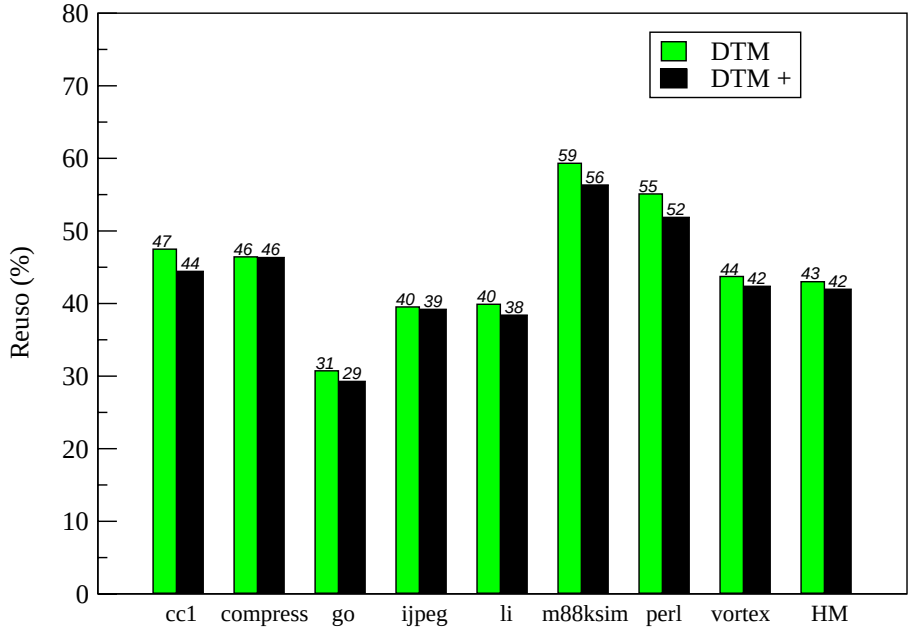


Figura 5.57: Variação no reuso, considerando uma configuração arquitetural mais agressiva, *SPECInt95*.

de 6% a 24% para os programas *go* e *compress* respectivamente, e apresentou média harmônica de 12.5%. Os programas *perl* e *vortex*, apresentaram ganhos de performance extremamente significativos 23% e 11% respectivamente, quando comparados aos mesmos na configuração conservadora *DTM*. Justificativas para o vortex, consideram o aumento do número de portas de acesso no cache de dados (53.46% das instruções dinâmicas deste programa correspondem a loads ou stores). Estas portas extras reduziram os gargalos causados por instruções de acesso à memória, pois estas não possuíam recursos para sua efetivação e não podiam prover regularmente os operandos para instruções ou traces testados como redundantes. Para o programa *perl*, como verificado anteriormente quando da avaliação do *DTM cache perf*, este se beneficia imensamente de acessos com sucesso ao cache de dados, e não poderia ser diferente na configuração *DTM+*, visto o aumento significativo do número de entradas nos caches *L1*.

Avaliando os resultados plotados para o *SPECFp95*, este apresentou reduções nos ganhos de performance (comparado com a configuração conservadora). Os ganhos de performance variaram de 1% a 15% para os programas *hydro2d* e *wave5* respectivamente, e apresentou média de 3.7%. O programa *wave5* apresentou um ganho

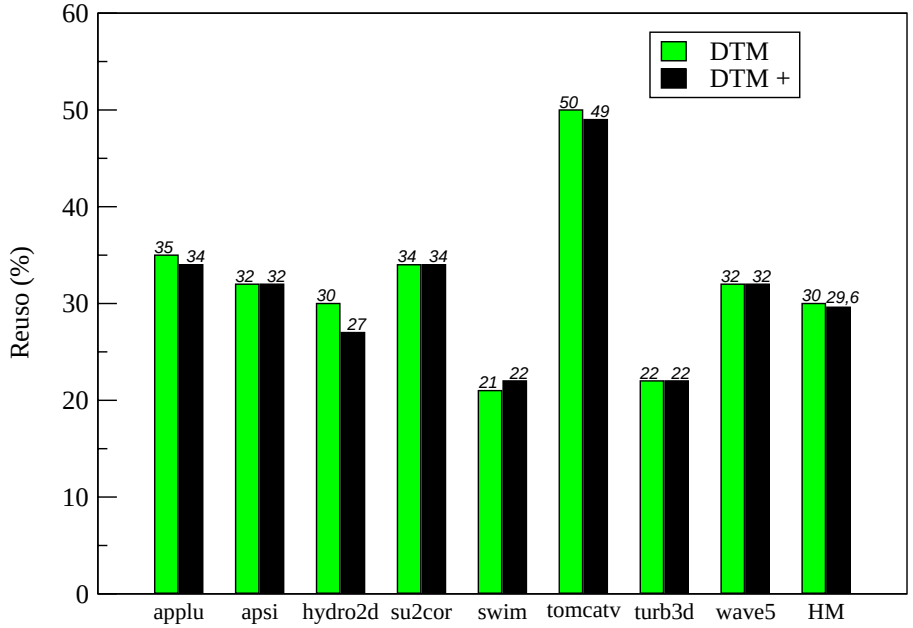


Figura 5.58: Variação no reuso, considerando uma configuração arquitetural mais agressiva, *SPECFp95*.

de performance significativo quando comparado ao seu desempenho na configuração conservadora *DTM*. Para este programa em particular, apesar do expressivo aumento do número de entradas no preditor de desvios, a exatidão da predição de desvios não variou muito da obtida pela configuração conservadora, principalmente para os desvios incondicionais indiretos (fonte significativa do ganho de performance). Para os outros programas o preditor de desvios atuou de forma decisiva, reduzindo os ganhos de performance para a configuração *DTM+*. O programa *hydro2d* apresentou uma redução expressiva do ganho de performance. Este comportamento foi decorrente do aumento do cache de instruções *L1* na configuração *DTM+*, visto que o efeito desta alteração proporcionou uma taxa de falha de 0% no acesso ao cache de instruções *L1*, e conseqüentemente produziu um resultado similar ao apresentado quando da avaliação da configuração *DTM cache perf* (o *hydro2d* possui seus ganhos de performance calcados na antecipação de instruções representadas nos traces, e que ainda não foram trazidas do cache de instruções em decorrência de acessos sem sucesso). Para o programa *swim*, o aumento do número de entradas no preditor de desvios não alterou a predição do alvo dos desvios incondicionais indiretos, e o ganho de performance para este programa foi ligeiramente maior (compensado pelo

efeito do aumento dos caches).

Analisando de forma global, a configuração $DTM+$ aumentou significativamente os ganhos de performance para o *SPECInt95* e os reduziu em mesma proporção para o *SPECFp95*. Entretanto, a redução média do ganho de performance para o *SPECFp95* foi decorrente principalmente do programa *hydro2d*.

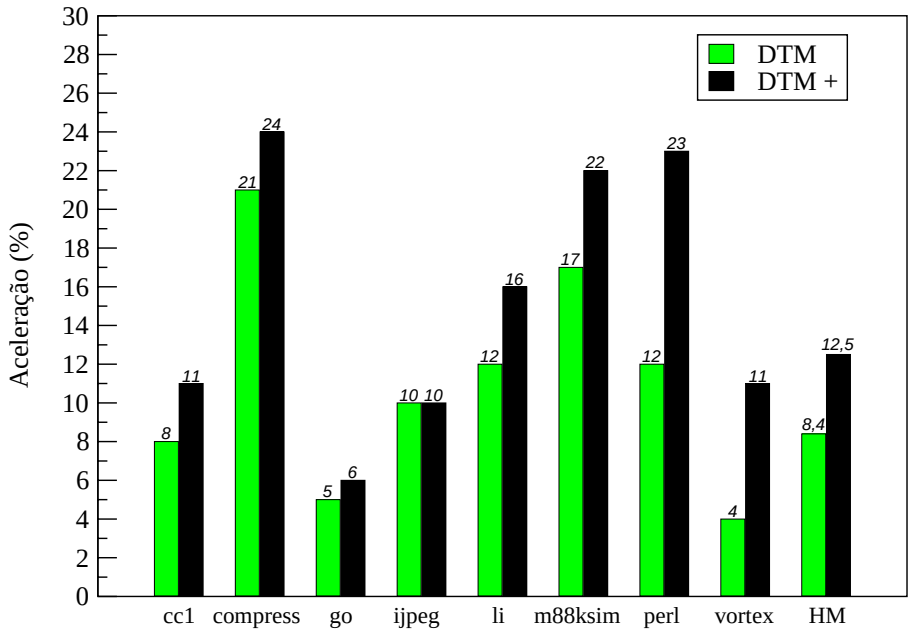


Figura 5.59: Variação nos ganhos de performance, considerando uma configuração arquitetural mais agressiva, *SPECInt95*.

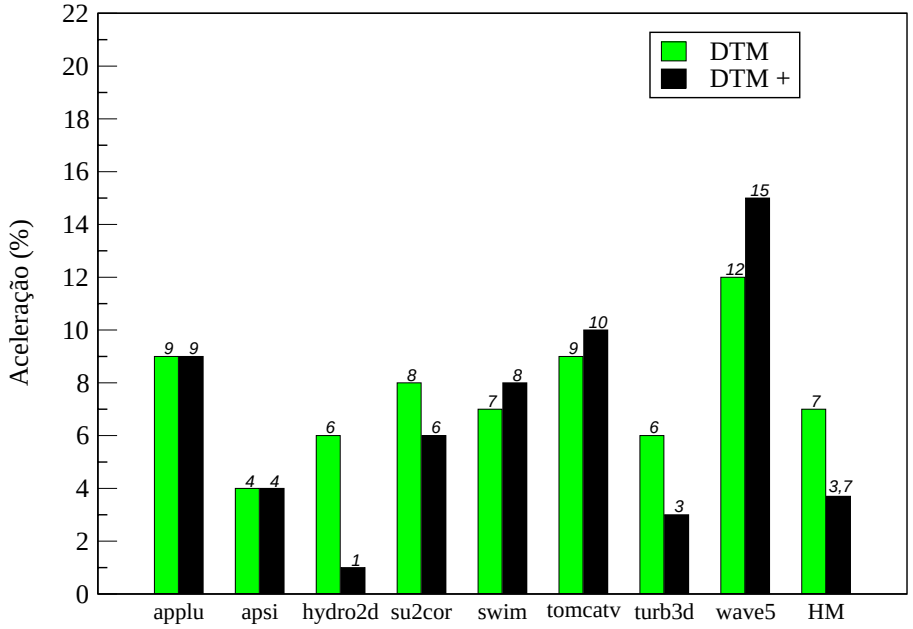


Figura 5.60: Variação nos ganhos de performance, considerando uma configuração arquitetural mais agressiva, *SPECFp95*.

Capítulo 6

Conclusões e Trabalhos Futuros

Neste trabalho de tese foi investigada, proposta e avaliada, uma variação da técnica de memorização com o objetivo de explorar dinamicamente o reuso de computações redundantes com granularidade em nível de traces e com exploração em nível de arquitetura de processador. As investigações efetuadas, verificaram a existência de redundância em nível de traces (seqüência de instruções dinâmicas). Foi proposto um mecanismo em nível de arquitetura de processador para, dinamicamente, identificar construtivamente, memorizar e reusar traces. A proposta estabelecida objetivou a eliminação da seleção estática das computações candidatas ao reuso e possíveis efeitos colaterais provocados pelo reuso. A partir desta proposta foi desenvolvido um mecanismo denominado *Dynamic Trace Memoization* (*DTM*), com a função de capturar e reusar traces sem auxílio do compilador ou alterações no conjunto de instruções básicas. O mecanismo desenvolvido baseia-se na seleção das instruções consideradas e utiliza-se de uma heurística para construir traces redundantes a partir destas instruções. O mecanismo desenvolvido foi incorporado a uma arquitetura superescalar e foi avaliado quanto à sua capacidade de identificar redundâncias e os ganhos de performance oriundos da exploração desta redundância.

Considerando os programas do *SPEC95*, comparações com outros mecanismos de reuso propostos, identificaram que o *DTM* é capaz de identificar um maior volume de redundância e explorá-la de forma mais eficiente para obtenção de ganhos de performance. Outras avaliações oriundas dos experimentos concluíram que:

- Comparativamente, o *DTM* reusa consideravelmente mais instruções que os demais esquemas de reuso propostos. Os percentuais de reuso identificado (média

harmônica) pelo *DTM*, S_{n+d} e *Block Reuse* foram de 43.6%, 19.2% e 22.2% respectivamente. Os percentuais de ganhos de performance (média harmônica) explorado pelo *DTM*, S_{n+d} e *Block Reuse* foram de 10.2%, 4.4% e 3.4% respectivamente;

• Considerando o *DTM* aplicado aos programas do *SPEC95*, foram identificados percentuais de reuso (média harmônica) de 43% e 30%, para os programas do *SPECInt95* e *SPECFp95* respectivamente, enquanto os percentuais de ganhos de performance (média harmônica) foram de 8.4% e 7%, para os programas do *SPECInt95* e *SPECFp95* respectivamente;

• O mecanismo *DTM* não requisita recursos adicionais de hardware ao processador superescalar que o incorpora. Foi constatado, por exemplo, que não são necessárias portas extras no arquivo de registradores para suportar as requisições para verificar a redundância de traces;

• A partir de observações sobre a composição dos traces redundantes, foram obtidas informações que determinam algumas de suas características e estabelecem alguns parâmetros arquiteturais do mecanismo (tamanho dos contextos de entrada e saída, número de portas de acesso à tabela de memorização de traces, etc...);

• O mecanismo *DTM* identifica uma grande quantidade de reuso e obtém bons ganhos de performance, mesmo quando incorporado a um processador com configurações ideais (preditor de desvios e caches perfeitos perfeitos). Para este caso, foram identificados percentuais de reuso (média harmônica) de 47.4% e 30.2%, para os programas do *SPECInt95* e *SPECFp95* respectivamente, enquanto os percentuais de ganhos de performance (média harmônica) foram de 7.6% e 2.7%, para os programas do *SPECInt95* e *SPECFp95* respectivamente;

• O potencial de redundância e os ganhos de performance obtidos quando todos os traces são reusados, são extremamente encorajadores e podem ser explorados pela conjunção de outros mecanismos atualmente em estudo (predição de valores). Para este caso, foram identificados percentuais de reuso (média harmônica) de 59% e 34%, para os programas do *SPECInt95* e *SPECFp95* respectivamente, enquanto os percentuais de ganhos de performance (média harmônica) foram de 19.3% e 10%, para os programas do *SPECInt95* e *SPECFp95* respectivamente;

• O desempenho do mecanismo *DTM* incorporado a processadores configura-

dos com diferentes larguras, forneceu indícios de que a exploração de reuso com granularidade em nível de traces, pode ser uma alternativa aos complexos projetos superescalares que exploram o aumento das larguras de cada estágio do pipeline para explorar paralelismo espacial e obtenção de ganhos de performance. Para os programas do *SPECInt95*, foram identificados percentuais de reuso (média harmônica) de 60%, 48% e 43%, para processadores com escalaridade 1, 2 e 4 respectivamente. Para os programas do *SPECFp95* foram identificados percentuais de reuso (média harmônica) de 37%, 32% e 32%, para processadores com escalaridade 1, 2 e 4 respectivamente. Com relação aos ganhos de performance obtidos: para os programas do *SPECInt95* foram observados percentuais de ganhos de performance (média harmônica) de 25%, 19% e 8.4%, para processadores com escalaridade 1, 2 e 4 respectivamente; para os programas do *SPECFp95* foram observados percentuais de ganhos de performance (média harmônica) de 9%, 13% e 7%, para processadores com escalaridade 1, 2 e 4 respectivamente;

• Considerando o *DTM* incorporado à um processador substrato com uma configuração arquitetural mais agressiva, foram identificados percentuais de reuso (média harmônica) de 42% e 29.8%, para os programas do *SPECInt95* e *SPECFp95* respectivamente, enquanto os percentuais de ganhos de performance (média harmônica) foram de 12.5% e 3.7%, para os programas do *SPECInt95* e *SPECFp95* respectivamente;

Por ser um trabalho inicial em exploração dinâmica de reuso em nível de arquitetura de processador e com granularidade em nível de traces, muitas questões não puderam ser cobertas, e muitos trabalhos decorrentes destas são alvo de pesquisas futuras. Entre as pesquisas a serem efetuadas, pode-se destacar as seguintes:

Avaliar alterações no modo de armazenamento, busca e política de atualização das tabelas de memorização do *DTM*. No mecanismo atual não existe limitação do número de diferentes instâncias de uma mesma instrução ou trace que podem ser armazenadas nas tabelas. Um próximo passo será verificar que alterações ocorrem quando o número de instâncias de uma mesma instrução ou trace são limitadas. Este procedimento possui como objetivo reduzir o grau de associatividade das tabelas de memorização, facilitando deste modo a lógica de acesso às tabelas (mapeamento direto com um limitado grau de associatividade).

Observou-se, durante o trabalho, a necessidade de se explorar o reuso de instruções de acesso à memória, apesar dos efeitos colaterais que estas introduzem. Para ampliar o percentual de reuso identificado e ganhos de performance obtidos pelo *DTM*, torna-se extremamente necessário investigar o efeito da incorporação de instruções de memória ao conjunto de instruções válidas do *DTM*, e o desenvolvimento de mecanismos que validem a consistência de valores armazenados na memória e nas tabelas de memorização. Trabalhos recentes [13, 10], podem prover uma direção para esta tarefa.

Investigar a viabilidade de desenvolvimento de mecanismos que identifiquem dinamicamente as instruções que produzem os valores de entrada dos traces. Estes mecanismos permitiriam identificar traces redundantes antecipadamente e, conseqüentemente, iriam aumentar o número de traces reusados. Estes mecanismos poderiam validar dinamicamente traces reusáveis, antes mesmo destes serem acessados.

Proposição e implementação de um mecanismo de predição de valores estendido à traces. Este mecanismo poderia permitir o reuso de traces especulativamente a partir de predições efetuadas sobre o contexto de entrada dos traces, principalmente para os casos em que estes encontram-se parcialmente instanciados. Esta proposta merece uma maior atenção em virtude do reduzido número de entradas de *Memo_Table_T*, do expressivo número de traces que não foram reusados por não possuírem seus operandos de entrada prontos e do expressivo ganho de performance identificado.

O compilador pode ter um papel extremamente importante no auxílio à exploração de reuso de traces. Ele, pode prover: a identificação de instruções que são invariantes (possuem um único valor a ela instanciado durante toda a execução do programa); a concentração seqüencial de instruções pertencentes ao domínio de instruções válidas no *DTM* (através de reorganização do código); a identificação e marcação de regiões com potencial de redundância; a identificação de instruções de memória que podem ser inseridas em traces; e transformações de código que possam expor uma maior quantidade de redundância. Extensões futuras deste trabalho de tese irão investigar a exploração de reuso efetuada pelo mecanismo *DTM* com suporte do compilador.

O reuso de traces pelo mecanismo *DTM* evita a execução e o tratamento de suporte (busca, despacho, escalonamento, entrega, etc...) de várias instruções du-

rante a execução de um programa. O reuso inibe a ativação de várias unidades de um processador, ou seja, várias unidades e recursos permanecem inativos ou são parcialmente ativados quando traces são reusados, principalmente quando os traces corrigem predições incorretas, evitando a avaliação de instruções em caminhos especulativamente incorretos. Esta característica pode ser explorada para implementação de processadores com baixo consumo de energia [8, 31]. Futuras investigações serão efetuadas para avaliar, em um processador incorporando o mecanismo *DTM*, a quantidade de elementos funcionais que permanecem inativos durante o reuso de traces. Um modelo de consumo de potência [15] será aplicado para medir a redução do consumo de energia do processador, considerando a quantidade total de inativações decorrentes do reuso de traces e considerando também o custo (em potência dissipada) adicionado pelo *DTM* para acesso às tabelas de memorização e ativação de seus estágios.

Capítulo 7

Bibliografia

[1] ADAN, T., CHANDY, K., DICKSON, J.R. "A Comparision of List Schedules for Parallel Processing Systems", *Communications of the ACM* v. 17, n. 12, pp. 685-690, December 1974.

[2] AHO, A., SETHI, R., ULLMAN J., *Compilers principles, techniques, and tools.* 1 ed. Reading, MA, Adison-Wesley, 1986.

[3] AGARWAL, V., HRISHKESH, M.S., KECKLER S.W., BURGER, D. "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures". In: *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 248-259, Vancouver, May 2000.

[4] APPEL, A.W., *Moderm Compiler Implementation in ML.* 1ed. New York, Cambridge University Press, 1998.

[5] ARVIND, CULLER, D.E., "Resource Requirements of Dataflow Program". In: *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 141-150, Honolulu, May 1988.

[6] ATKINS, D.E. "Higher-radix division using estimates of the divisor and partial reminders", *IEEE Transactions on Computers* v. 17, n. 10, pp. 925-934, October 1968.

[7] AUSTIM, T., SOHI, G.S. "Dynamic Dependency Analysis of Ordinary Programas". In: *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 342-351, Melbourne, May 1992.

[8] AZAM, M., FRANZON,P., LIU, W., CONTE, T. "Low Power Data Processing by Elimination of Redundant Computations". In: *Proceedings of the International Symposium on Low-Power Eletronics and Design*, pp. 259-264, August 1997.

[9] BACON, D.F., GRAHAN, S.L., SHARP, O.L. "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, v. 26, n. 2, pp. 345-420, December 1994.

[10] BELL, G., LEPAK,K., LIPASTI, M. "Characterization of Silent Stores". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 133-142, Philadelphia, October 2000.

[11] BLUM, M., WASSERMAN, H. "Reflections on the Pentium Division Bug", *IEEE Transactions on Computers* v. 45, n. 4, pp. 385-393, April 1996.

[12] BODIK, R., GUPTA, R., SOFFA M.L. "Complete removal of redundant expressions". In: *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1-14, Montreal, June 1998.

[13] BODIK, R., GUPTA, R., SOFFA M.L. "Load-Reuse Analysis: Design and Evaluation". In: *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pp. 64-76, Atlanta, May 1999.

[14] BRIGGS P., COOPER, K. "Effective Partial Redundancy Elimination". In: *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pp. 159-170, Orlando, June 1994.

[15] BROOKS, D., TIWARI, V., MARTONOSI, M. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations". In: *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 83-94, Vancouver, May 2000.

[16] 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.

[17] CALDER, B., FELLER, P., EUSTACE, A. "Value Profiling". In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 259-269, Melbourne, December 1997.

[18] CITRON, D., FEITELSON, D., RUDOLPH, L. "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units". In: *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 252-261, San Jose, October 1998.

[19] CONNORS, D.A., HWU, W.W. "Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results". In: *Proceedings of the 32th International Symposium on Microarchitecture*, pp. 158-169, Haifa, November 1999.

[20] da COSTA, A.T., FRANÇA, F.M.G. *The Reuse Potencial of Trace Memoization*, Technical Report ES-498/99, COPPE/UFRJ, Rio de Janeiro, May 1999.

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

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

[23] da COSTA, A.T., FRANÇA, F.M.G., CHAVES, E.M.F. "The Dynamic Trace Memoization Reuse Technique". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 92-99, Philadelphia, October 2000.

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

[25] DEC, *Alpha Architectures Handbook*. 1 ed. Maynard, MA, Digital Equipment Corporation, 1996

[26] FISHER, J.A. "Very Long Instruction Word Architectures and the ELI-512". In: *Proceedings of the 10th International Symposium on Computer Architecture*, pages 140-150, an Diego, June 1983.

[27] FRIENDLY, D.H., PATEL, S.J., PATT, Y.N."Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors". In: *Proceedings of the 31th International Symposium on Microarchitecture*, pp. 173-181, Dallas, November 1998.

[28] GRANT B., MOCK, M., PHILIPOSE M., CHAMBERS C., EGGERS S. "Anotation-Directed Run-Time Specialization in C". In: *Proceedings of the ACM/SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 163-178, Amsterdam, June 97.

[29] GABBAY, F., MENDELSON, A. *Speculative Execution based on Value Prediction*, Technical Report EE-TR 1080, Technion - Israel Institute of Technology, November 1996.

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

[31] GONZALEZ, R., HOROWITZ, M. "Energy Dissipation in General Purpose Microprocessors", *IEEE Journal of Solid State Circuits* v. 31, n. 9, pp. 1277-1284, September 1996.

[32] HARBISON, S.P. "An Architectural Alternative to Optimizing Compilers". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 57-65, Palo Alto, March 1982.

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

[34] HENNESSY, J., PATTERSON, D., *Computer Architecture A Quantitative Approach.* 1 ed. San Mateo, CA, Morgan Kaufmann Publishers, 1995.

[35] HUANG, J., LILJA, D.J. "Exploiting Basic Block Value Locality with Block Reuse". In: *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pp. 106-114, Orlando, January 1999.

[36] HUANG, J., LILJA, D.J. "Exploring Sub-Block Value Reuse for Superscalar Processors". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pp. 100-107, Philadelphia, October 2000.

[37] IBM, "IBM Regains Performance Lead with POWER2", *Microprocessor Report* v. 7, n. 13, October 1993.

[38] IBM, MOTOROLA. *PowerPC 750 RISC Microprocessor*, Technical Summary Order Number MPC750/D (also available at http://www.chips.ibm.com/products/ppc/), 1997.

[39] JACOBSON Q., ROTENBERG, E., SMITH, J. E. "Path-Based Next Trace Prediction". In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 14-23, Melbourne, December 1997.

[40] JOHNSON, M., *Superscalar Microprocessor Design*, 1 ed, New Jersey, Prentice Hall, 1991.

[41] JOLLY, R. " A 9-ns, 1.4 Gigabyte/s, 17-Ported CMOS Register File", *IEEE Journal of Solid State Circuits* v. 26, n. 10, pp. 1407-1412, October 1991.

[42] KNOBLOCK, T.B., RUF, E. "Data Specialization". In: *Proceedings of the ACM/SIGPLAN Conference on Programming Language Design and Implementation*, pp. 149-159, Philadelphia, June 1996.

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

[44] LIPASTI, M.H., SHEN, J.P. "Exceeding the Dataflow Limit Via Value Prediction". In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 226-237, Paris, December 1996.

[45] LIPASTI, M.H., WILKERSON, C.B., SHEN, J.P. "Value Locality and Load Value Prediction". In: *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138-147, Cambridge,MA, September 1996.

[46] LIU, Y.A., TEITELBAUM, T. "Caching Intermediate Results for Program Improvement". In: *Proceedings of the ACM/SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 190-201, La Jolla, June 1995.

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

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

[49] MOLINA, C., GONZALEZ, A., TUBELLA, J. "Dynamic Removal of Redundant Computations". In: *Proceedings of the International Conference on Supercomputing*, pp. 474-481, Rhodes, July 1999.

[50] OBERMAN, S., FLYNN, M. "Reducing Division Latency with Reciprocal Caches", *Reliable Computing* v. 2, n. 2, pp. 147-153, April 1996.

[51] PALACHARLA, S., JOUPPI, N.P., SMITH, J.E. *Quantifying the Complexity of Superscalar Processors*, Technical Report CS-TR-96-1328, University of Wisconsin-Madison, Wisconsin, November 1996.

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

[53] RICHARDSON, S.E. *Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation*, Technical Report SMLI TR-92-1, Sun Microsystems Laboratories, California, Sept.1992.

[54] RISEMAN, E.M., FOSTER, C.C. "The Inhibition of Potential Parallelism by Conditional Jumps", *IEEE Transations on Computers* v. 21, n.12, pp. 1405-1411, December 1972.

[55] RIVERS, J.A., TYSON, G.S., DAVIDSON, E.S., AUSTIM, T.M. "On High-Bandwidth Data Cache Design for Multi-Issue Processor". In: *Proceedings of the 30th International Symposium on Microarchitecture*, pp. 46-56, Melbourne, December 1997.

[56] ROTENBERG, E., BENNETT, S., SMITH, J. E. "Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching". In: *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24-34, Paris, December 1996.

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

[58] SAZEIDES,.Y., VASSILIADIS, S., SMITH, J.E. "The Performance Potencial of Data Dependence Speculation and Collapsing". In: *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 238-247, Paris, December 1996

[59] SMITH, J.E. "A Study of Branch Prediction Strategies". In: *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135-148, May 1981.

[60] SMITH, J. E., PLESZKUN, A. "Implementing Precise Interrupts in Pipelined Processors", *IEEE Transactions on Computers* v. 37, n. 5, pp. 562-573, May 1988.

[61] SMITH, J.E., SOHI, G. S. "The Microarchitecture of Superscalar Processors", *Proceedings of the IEEE Transactions on Computers* v. 83, n. 12, pp. 1609-1624, December 1995.

[62] SOHI, G.S. "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers", *IEEE Transactions on Computers* v.39 n. 3, pp. 349-359, March 1990.

[63] SODANI, A., SOHI, G.S. "Dynamic Instruction Reuse". In: *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205, Denver,

July 1997.

[64] 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.

[65] SODANI, A., SOHI, G.S. "Understanding Differences Between Value Prediction and Instruction Reuse". In: *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pp. 205-215, Dallas, December 1998.

[66] SHRIVER, B., SMITH, B., *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. 1 ed. Los Alamitos, CA, IEEE Computer Society Press, 1998.

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

[68] WILLIAMS, T., PATKAR, N., SHEN, G. "SPARC64: A 64-b 64-Active Instruction Out-of-Order-Execution Processor", *IEEE Journal of Solid State Circuits* v. 30, n. 11, pp. 1215-1226, November 1995.

[69] WOLFE, A., FRITS, J., DUTTA, S., FERNANDES, E.S.T. "Datapath Design for a VLIW Video Signal Processor". In: *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp. 24-35, San Antonio, February 1997.

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

[71] ZHANG, C., ZHANG, X., YAN, Y. "Two Fast and High-Associativity Cache Schemes", *IEEE Micro* v. 17, n. 5, pp.40-50, October 1997.