Avaliando DTM em Arquiteturas Superescalares Configuradas com Diferentes Larguras

Amarildo T. da Costa[1,2], Felipe M. G. França[2], Eliseu M. C. Filho[2]
[1] Dept. Engenharia Elétrica
IME - Instituto Militar de Engenharia
Pça Gal. Tibúrcio 80
22290-270 Rio de Janeiro, Brasil
[2] Dept. Engenharia de Sistemas e Computação
COPPE/Universidade Federal do Rio de Janeiro
Caixa Postal 68511
21945-970 Rio de Janeiro, Brasil
Comentários e sugestões para: {amarildo,felipe,eliseu}@cos.ufrj.br

Resumo—

Este trabalho avalia a exploração de redundância em nível de traços (seqüências de instruções dinâmicas) aplicada a processadores superescalares com diferentes larguras. A redundância existente em programas foi explorada através de um mecanismo de reuso denominado *Dynamic Trace Memoization* (*DTM*). Simulações considerando processadores configurados com diferentes larguras e incorporando o mecanismo *DTM*, identificaram para os programas do *SPEC95*: percentuais de reuso variando de 28% a 60% (média harmônica); percentuais de ganhos de performance variando de 6.3% a 25% (média harmônica); e que um processador superescalar com largura 4 e incorporando o mecanismo *DTM* produz ganhos de performance sobre o mesmo processador superescalar base com largura 8. Este último resultado fornece fortes indícios de que a exploração de redundância em nível de traços, apresenta-se como uma alternativa viável à opção de se aumentar as larguras dos processadores superescalares para a obtenção de um maior número de instruções executadas por ciclo de clock.

Palavras-chave— **Reuso de Traces, Memorização, Reuso de Instruções, Processadores Superescalares.**

Abstract—

This work evaluates the exploration of redundancy at the trace level (sequences of dynamic instructions) applied to superscalar processors with different widths. The existing redundancy in programs was explored through a reuse mechanism called *Dynamic Trace Memoization* (*DTM*). Simulations considering processors configured with different widths and incorporating the *DTM* mechanism, had identified for the *SPEC95* benchmarks: reuse percents from 28% to 60% (harmonic mean) ; speedup percents from 6.3% to 25% (harmonic mean); and that a superscalar processor with width 4 and incorporating the *DTM* mechanism, outperform in performance the same base superscalar processor with width 8. This last result supplies strong indications of that the redundancy exploration at the trace level is a viable alternative to the option of magnifying the widths of superscalar processors to increase the number of executed instructions per clock cycle.

Keywords— **Trace Reuse, Memoization, Instruction Reuse, Superscalar Processor.**

I. Introdução

Projetos de processadores superescalares mais velozes, procuram executar um maior número de instruções por ciclo de clock (*ipc*). Este objetivo tem sido alcançado principalmente através do aumento da largura do processador (replicação dos estágios), o que envolve um alto grau de complexidade e custos elevados [PAL 96]. Aumentar a largura de um processador equivale a aumentar a quantidade de instruções tratadas pelos estágios deste processador, de modo a alcançar um maior *ipc*. A realização deste objetivo esbarra principalmente em dificuldades relativas ao tratamento de instruções de controle (fluxo de execução) e na serialização imposta pelas dependências verdadeiras entre instruções.

Em contrapartida, recentes pesquisas [SOD 97, HUA 99, COS 99, GON 99], revelaram a existência de uma grande quantidade de computações redundantes[1] em programas. Estas computações redundantes quando reusadas: reduzem o número de instruções executadas; desconsideram a serialização imposta pelas cadeias de instruções dependentes; reduzem a contenção de recursos funcionais; e corrigem os efeitos provocados por desvios preditos incorretamente. Deste modo, estes mecanismos podem contribuir para eliminar algumas das restrições impostas à exploração de um maior valor de *ipc* em processadores superescalares.

Neste trabalho, serão efetuadas simulações considerando a variação das larguras de um processador superescalar incorporando um mecanismo de reuso denominado *Dynamic Trace Memoization* (*DTM*) [COS 00]. Os resultados determinarão o desempenho do mecanismo *DTM* para as diferentes configurações do processador e, a partir destes, estaremos aptos a identificar e comparar diferentes alternativas para acelerar o desempenho de processadores superescalares.

Este trabalho está organizado da seguinte forma: A seção II apresenta e descreve o mecanismo *DTM* e seu funcionamento. Na seção III é descrita a incorporação do mecanismo *DTM* em uma microarquitetura superescalar. A seção IV revisa os trabalhos relacionados. A seção V descreve o ambiente experimental utilizado. A seção VI descreve e discute os resultados obtidos na simulação. Finalmente na seção VII, são expostas as principais conclusões e são identificados os

[1] Computações redundantes englobam as diversas granularidades: instruções, blocos básicos, traços, funções, etc...

trabalhos futuros desta pesquisa.

II. Dynamic Trace Memoization

Dynamic Trace Memoization - (*DTM*) é um mecanismo que explora dinamicamente o reuso de computações redundantes com granularidade em nível de traços e em nível de arquitetura de processador.

Um *traço* é uma seqüência dinâmica de instruções executadas por um programa. Um traço é redundante, se composto por instruções redundantes. O *domínio de instruções válidas para o DTM* é composto pelo subconjunto de instruções do processador alvo que são selecionadas como candidatas para compor os traços considerados. Considerando o conjunto de instruções do processador MIPS ISA (avaliado neste trabalho), apenas as instruções `LOAD`, `STORE` e `FLOATING-POINT`, não pertencerão ao *domínio de instruções válidas para o DTM*. Razões para a exclusão destas instruções são: instruções de acesso à memória [2] (`LOAD` e `STORE`) podem gerar efeitos colaterais ao serem reusadas (embora possam reusar o mesmo endereço de memória sucessivas vezes, não é garantido que o valor armazenado no endereço é igualmente mantido inalterado); e instruções de ponto flutuante (`FLOATING-POINT`) não apresentam boa localidade de valores [LIP 96];

O *contexto de entrada* de um traço corresponde ao conjunto de seus operandos de entrada e valores a eles instanciados, enquanto o *contexto de saída* de um traço corresponde ao conjunto de seus operandos de saída e valores a eles instanciados, sendo estes, resultados produzidos por instruções do traço.

Um traço é redundante, se o seu contexto de entrada é idêntico ao contexto de entrada do mesmo traço já observado e memorizado. Na seqüência, será explicado como traços são construídos, memorizados e reusados.

A. Construção de Traços Redundantes

Traços redundantes em *DTM*, são construídos a partir da identificação de instruções dinâmicas redundantes e pertencentes ao domínio de instruções válidas do *DTM*. Para tal propósito, o *DTM* inclui uma *tabela de memorização global* denominada *Memo_Table_G*, que armazena em tempo de execução as instruções dinâmicas que pertencem ao domínio de instruções válidas do *DTM*. Cada entrada de *Memo_Table_G* armazena uma instrução dinâmica (instanciada), e possui o formato apresentado na Figura 1. O campo *pc* armazena o endereço de memória da instrução, os campos *sv1* e *sv2* armazenam os valores instanciados aos operandos de entrada e o campo *res/npc* armazena o resultado de uma instrução aritmética/lógica ou o endereço alvo de uma instrução de controle. Os bits *jmp* e *brc* identificam se a instrução é um jump ou branch respectivamente, e o bit *btk* indica se o branch foi tomado ou não.

[2] Cálculos de endereços de memória são reusados pelo mecanismo *DTM*

32	1	1	1	32	32	32
pc	jmp	brc	btk	sv1	sv2	res/npc

Figura 1. Estrutura de uma entrada de *Memo_Table_G*.

Inicialmente, o *DTM* verifica para cada instrução dinâmica, se esta pertence ao seu domínio de instruções válidas. Se a instrução é inválida, então o *DTM* rotula a instrução como "não redundante", e não a insere em *Memo_Table_G*. Caso contrário, o *DTM* busca por uma entrada em *Memo_Table_G* que possua os campos *pc*,*sv1* e *sv2* com os mesmos valores instanciados ao endereço da instrução e os valores correntes dos operandos. Se a entrada pesquisada não existir, então o *DTM* rotula a instrução como "não redundante", e a insere em *Memo_Table_G*. Se a entrada pesquisada for encontrada, então *DTM* rotula a instrução como "redundante", e não a insere em *Memo_Table_G*.

Traços são construídos a partir de instruções rotuladas como redundantes. Se a instrução estiver rotulada como não redundante, então o *DTM* finaliza qualquer traço que esteja em construção. Caso contrário, para um traço em construção, a informação de contexto é atualizada (considerando a instrução redundante) ou é iniciada a construção de um novo traço. A Figura 2 esboça o procedimento para construção de traços a partir de instruções rotuladas como redundantes em *Memo_Table_G*.

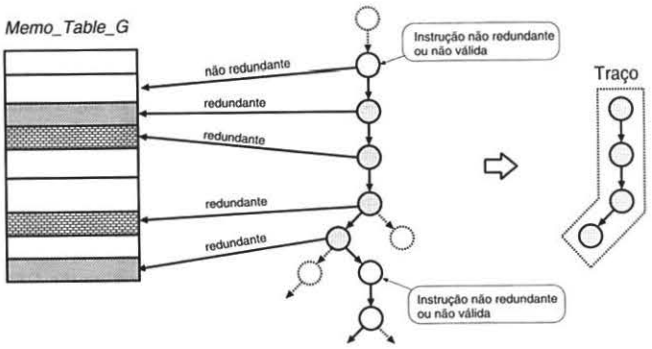

Figura 2. Construção de um traço.

A Figura 3 apresenta o formato das entradas de *Memo_Table_T*. O campo *pc* armazena o endereço da instrução inicial do traço. O campo *npc* especifica o endereço da próxima instrução a ser executada para o caso em que o traço seja reusado, este é preenchido com o endereço da próxima instrução a ser executada após a última instrução do traço. O campos *icv* armazenam os valores do contexto de entrada do traço, enquanto os campos *icr* identificam os correspondentes

registradores fonte. Os campos *ocv* armazenam os valores do contexto de saída, enquanto os respectivos registradores de destino são especificados pelos campos *ocr*. Cada bit do campo *bmsk* indica a presença de uma instrução de desvio condicional (branch) no traço, enquanto o correspondente bit no campo *btk* indica se o correspondente branch foi tomado ou não. O *DTM* finaliza a construção de um traço sempre uma instrução não redundante é encontrada. Os valores N e B indicam respectivamente o número máximo de elementos nos contextos de entrada e saída e o número máximo de desvios permitidos. É importante ressaltar que o campo *npc* engloga todas as mudanças de fluxo inclusas no traço.

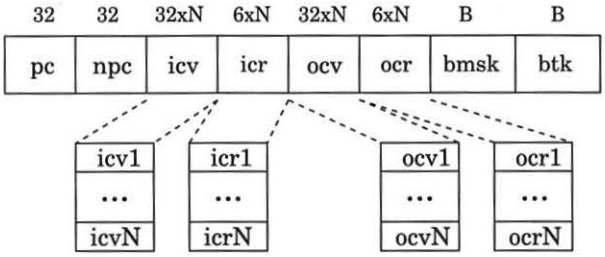

Figura 3. Estrutura de uma entrada de *Memo_Table_T*.

B. Reusando Instruções e Traços

Concorrentemente à construção de traços, o *DTM* procura por instruções redundantes e traços que possam ser reusados. Para detectar um traço redundante, o *DTM* inicialmente verifica para cada instrução dinâmica, se esta, é a instrução inicial de um traço previamente armazenado em *Memo_Table_T*. Esta verificação é efetuada, pesquisando-se em *Memo_Table_T* por uma entrada possuindo o campo *pc* idêntico ao endereço da instrução dinâmica. Para as entradas de *Memo_Table_T* que satisfaçam a comparação com o campo *pc*, serão comparados os conteúdos correntes dos registradores do processador apontados pelos campos *icr*, contra os valores do contexto de entrada armazenados nos correspondentes campos *icv*. O traço será redundante se for identificada alguma entrada de *Memo_Table_T* (dentre as previamente selecionadas) que satisfaça a comparação de contexto de entrada configurada acima. Para detectar uma instrução redundante, o *DTM* efetua uma busca em *Memo_Table_G*, procurando por entradas que possuam o campo *pc* idêntico ao endereço da instrução dinâmica corrente. Para as entradas selecionadas pela comparação do campo *pc*, serão pesquisados os campos *sv1* e *sv2* de *Memo_Table_G* que sejam idênticos aos valores dos operandos da instrução corrente. A instrução será redundante se for identificada alguma entrada de *Memo_Table_G* (dentre as previamente selecionadas) que satisfaça a comparação de contexto de entrada configurada acima.

É imediato observar que as operações para detectar instruções redundantes e traços redundantes são análogas, a única diferença refere-se a estrutura acessada. Instruções redundantes são detectadas através de *Memo_Table_G*, enquanto traços 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 traço redundante é encontrado.

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/npc* da correspondente entrada selecionada em *Memo_Table_G*. No caso de instruções de desvio redundantes, os campos *res/npc* e *btk* 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 traço 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 traço. Em adição, o *DTM* atualiza o estado do preditor de desvios usando a informação armazenada nos campos *bmsk* e *btk*.

III. Uma Microarquitetura Incorporando o Mecanismo *DTM*

Para a incorporação do mecanismo *DTM*, será considerada uma microarquitetura substrato correntemente encontrada em processadores superescalares. Exceto por poucas diferenças, a microarquitetura escolhida é similar à implementada no AMD K6-III [SHI 98].

O *Estágio de Busca de Instruções* busca instruções no cache de instruções e as insere no *Buffer de Instruções*.

O *Estágio de Decodificação* obtém instruções do *Buffer de Instruções*, efetua sua decodificação, leitura de seus operandos (e busy bits) de entrada no arquivo de registradores e as insere no *Buffer de Emissão*. O *Buffer de Emissão* é logicamente organizado como uma fila circular que armazena as instruções a serem despachadas para a execução. Este provê um conjunto de estações de reservas centralizadas e a funcionalidade de um *Buffer de Reordenação* para suportar interrupções precisas.

O *Estágio de Emissão* percorre as entradas do *Buffer de Emissão* procurando por instruções aptas para execução (dependendo da disponibilidade de seus operandos de entrada). Instruções aptas são despachadas para *Estágio de Execução*,

caso existam unidades funcionais disponíveis.

O *Estágio de Execução* efetua a execução das instruções, independente de sua ordem no *Buffer de Emissão*. Quando uma instrução é executada, o resultado produzido é repassado para o *Estágio de Emissão*. Este, escreve o resultado produzido na respectiva entrada do *Buffer de Emissão* ocupada pela instrução já executada, e repassa o resultado para possíveis instruções dependentes (disponibilizando novas instruções para a execução). A partir do exposto, a instrução executada é completada e disponibilizada para o *Estágio de Entrega*, que as retira em ordem do *Buffer de Emissão* e efetua a atualização do arquivo de registradores.

Assumindo esta microarquitetura substrato (base), o mecanismo *DTM* será organizado em três estágios em pipeline: *DS1*, *DS2* and *DS3*; que operam em paralelo com os *Estágio de Busca de Instruções*, *Estágio de Decodificação* e *Estágio de Entrega*. A Figura 4 apresenta o esboço da microarquitetura substrato incorporando o mecanismo *DTM*, suas tabelas de memorização e seus estágios. Adiante, será fornecida, uma breve descrição do funcionamento dos estágios do *DTM*.

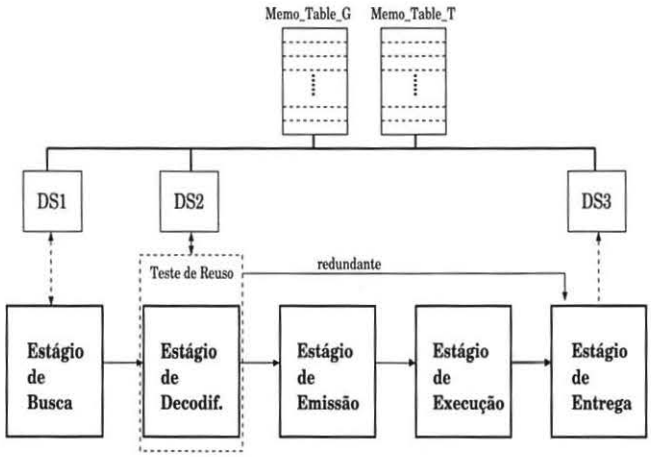

Figura 4. Microarquitetura incorporando o *DTM*.

A. *Operações relacionadas à construção de Traços*

O *Estágio DS1* seleciona as entradas de *Memo_Table_G* que possuam o campo *pc* idêntico ao endereço de cada instrução acessada pelo *Estágio Busca*.

O *Estágio DS2* captura os valores dos operandos lidos (das instruções) pelo *Estágio de Decodificação* e compara aos respectivos campos *sv1* e *sv2* das entradas previamente selecionadas em *Memo_Table_G* pelo *Estágio DS1*. Se a comparação identificar a igualdade de valores, então o *Estágio DS2* indicará ao *Estágio de Decodificação* que a instrução correspondente é redundante.

O *Estágio DS3* possui uma estrutura denominada *buffer temporário* (com composição idêntica à uma entrada de *Memo_Table_T*), e que armazena as instruções dinâmicas redundantes para composição dos traços. O *Estágio DS3* captura os identificadores dos registradores fonte (operandos de entrada), os valores instanciados aos operandos de entrada, o identificador do registrador de destino (operando de saída) e o resultado de cada instrução entregue pelo *Estágio de Entrega*. Se a instrução é redundante, o *Estágio DS3* preenche os apropriados campos do *buffer temporário*. Se a instrução não é redundante, então o *Estágio DS3* finaliza o traço em construção e transfere a informação do *buffer temporário* para alguma entrada de *Memo_Table_T*. Por capturar somente as instruções que são entregues pelo *Estágio de Entrega*, é garantido que os traços e instruções redundantes correspondem somente aos caminhos de execução corretos, ou seja, não serão especulativos.

B. *Operações relacionadas ao reuso de Traços ou Instruções*

O *Estágio DS1* é responsável pela seleção prévia via endereço da instrução corrente, de traços ou instruções em *Memo_Table_T* e *Memo_Table_G* respectivamente,

O *Estágio DS2* é responsável pela identificação de traços ou instruções redundantes, sobre as entradas selecionadas pelo *Estágio DS1*.

Para cada instrução redundante identificada, o *Estágio de Decodificação* irá inserir no *Buffer de Emissão* a instrução rotulada como redundante e seu respectivo resultado obtido da correspondente entrada de *Memo_Table_G*. Se um traço redundante for identificado, então o *Estágio de Decodificação* irá inserir no *Buffer de Emissão* os pares <*ocr*,*ocv*> (da correspondente entrada de *Memo_Table_T*) que atualizarão o arquivo de registradores. Em ambos os casos, as entradas inseridas do *Buffer de Emissão* não serão consideradas para execução, porém os resultados serão reenviados para outras instruções que os requisitem. O *Estágio de Entrega* tratará estas entradas normalmente para atualização do estado arquitetural do processador.

IV. Trabalhos Relacionados

Em [SOD 97], Sodani e Sohi introduzem o conceito de *dynamic instruction reuse* e descrevem três esquemas de reuso nomeados S_v, S_n e S_{n+d}. Todos os esquemas são baseados em uma estrutura denominada *Reuse Buffer* (*RB*), que salva as informações de operandos e resultados de instruções dinâmicas, de modo a permitir que as mesmas sejam reusadas quando identificadas novamente. Os esquemas S_v e S_n reusam instruções simples, enquanto o esquema S_{n+d} é capaz de reusar (de forma limitada) cadeias de instruções dependentes.

Em [HUA 99], Huang e Lilja introduzem a noção de *basic block value locality* e apresentam o mecanismo *block reuse*. A idéia é explorar o reuso considerando uma granulari-

dade maior (blocos básicos). O *block reuse* emprega uma estrutura denominada *Block History Buffer (BHB)* para armazenar dinamicamente o conjunto de entradas e saídas de blocos básicos. Posteriormente em [HUA 00], mantendo-se a mesma estrutura de armazenamento do *block reuse*, foi apresentada uma nova proposta que procura explorar o reuso de *sub-blocks*, onde o bloco básico é particionado em sub-blocos menores através de heurísticas, de modo a aumentar as oportunidades de reuso.

Em [GON 99], Gonzalez *et al.* é estudado o potencial de reuso de valores em nível de traços. Neste trabalho foi empregada uma estrutura denominada *Reuse Trace Memory (RTM)*, que armazena os traços determinados idealmente, e os reusa quando identificados. Entretanto, não foi estabelecida nenhuma proposta para sua incorporação em uma micro-arquitetura real.

Uma diferença chave do *DTM* com relação aos mecanismos de reuso mencionados, recai na exclusão de instruções de acesso à memória. Temos preferido não considerar estas instruções, para evitar a complexidade de se manter as tabelas consistentes em relação a memória, pois para tal propósito seria necessário detectar todas as instruções que escrevem na memória (`STORE`) e atualizar as entradas das tabelas que fazem referência a posição alterada.

V. Ambiente Experimental

Para a realização dos experimentos foi utilizado o simulador `sim-outorder` do *SimpleScalar Tool Set, Version 2.0* [BUR 97], modificado para suportar a incorporação o mecanismo *Dynamic Trace Memoization - DTM*.

Como benchmarks, foram usados os programas do *SPEC95Int95* e *SPECFp95*. Estes foram compilados usando o compilador C, fornecido pelo SimpleScalar Tool Set (`gcc-2.6.3`) e com a opção de otimização *-O3* ativada. Os programas `perl` e `vortex` foram executados para 300 milhões de instruções, os programas `hydro2d`, `turb3d` e `wave5` foram executados para 500 milhões de instruções, enquanto os programas restantes foram executados completamente.

O efeito sobre a performance foi medido como aceleração, e é dada pela razão ipc_{DTM}/ipc_{BASE}, onde ipc_{DTM} representa o número médio de instruções executadas por ciclo para a arquitetura incorporando o *DTM*, e ipc_{BASE} representa o número médio de instruções executadas por ciclo para a arquitetura base (original sem o *DTM*). O reuso é dado pela razão N_r/N_t, onde N_r é o número de instruções reusados (individualmente ou como parte de um traço) e N_t representa o total de instruções executadas. Cálculo de endereços reusados em instruções de acesso à memória são contabilizados como instruções simples do total de instruções executadas N_t.

Serão avaliadas comparativamente, quatro configurações do mesmo processador substrato e suas respectivas versões incorporando o mecanismo *DTM*. A tabela I 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 traços 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 pelas configurações *w=4* e *w=8*. Enquanto para configuração *w=1*, este buffer foi reduzido para se aproximar de uma configuração escalar real. Outros elementos da arquitetura: cache de instruções com 16Kbytes, cache de dados com 16Kbytes e o preditor de desvios por correlação e com 2k entradas, foram mantidos inalterados para todas as configurações. As tabelas de memorização foram configuradas com 512 entradas para *Memo_Table_G* e 4672 entradas para *Memo_Table_T* [COS 00], ambas com política de gerenciamento *LRU*. Os valores N e B definidos anteriormente, possuirão os valores 4 e 6 respectivamente.

Tabela I
CONFIGURAÇÕES DO PROCESSADOR PARA AS DIFERENTES LARGURAS.

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

VI. Resultados

A. Reuso Identificado

Para todos os programas avaliados em todas as configurações arquiteturais do processador considerado, foram identificados expressivos percentuais de reuso decorrentes da aplicação do mecanismo *DTM*.

Nas Figuras 5 e 6, são apresentados os percentuais de reuso obtidos para cada configuração. Analisando os valores plotados para os programas do *SPECInt95*, nestes, foram obtidos em média harmônica, percentuais de reuso de 60%, 48%, 43% e 41%, para as configurações *w=1*, *w=2*, *w=4* e *w=8* respectivamente. Observa-se que o percentual de reuso explorado é decrescente na medida em que são crescentes as larguras do processador. Esta característica é decorrente da restrição imposta pela indisponi-

bilidade dos operandos do contexto de entrada dos traços e instruções redundantes, e que ocorre principalmente na configuração *w=4* e *w=8*. Isto se justifica, pois o traços e instruções simples são verificados como sendo redundantes, somente para caso em que seus operandos de entrada estiverem instanciados à valores válidos e apenas enquanto presentes no *Estágio de Decodificação*. Assim sendo, esta consideração pode penalizar mais acentuadamente os processadores com configurações arquiteturais que tratam múltiplas instruções. Estas restrições são praticamente anuladas quando a configuração *w=1* é utilizada. Análise análoga para os programas do *SPECFp95*, identificaram em média, percentuais de reuso de 37%, 32%, 32% e 28%, para as configurações *w=1*, *w=2*, *w=4* e *w=8* respectivamente. As mesmas considerações já expostas para o *SPECInt95* são válidas para o *SPECFp95*, entretanto as configurações *w=2* e *w=4* apresentaram o mesmo percentual de reuso.

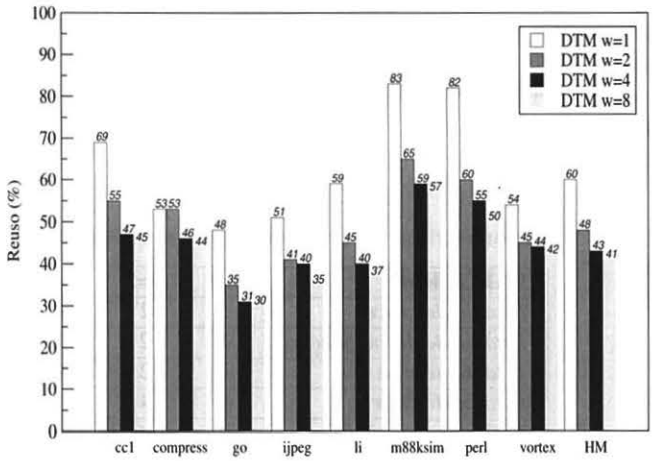

Figura 5. Reuso explorado pelo *DTM* em processadores com diferentes larguras, *SPECInt95*

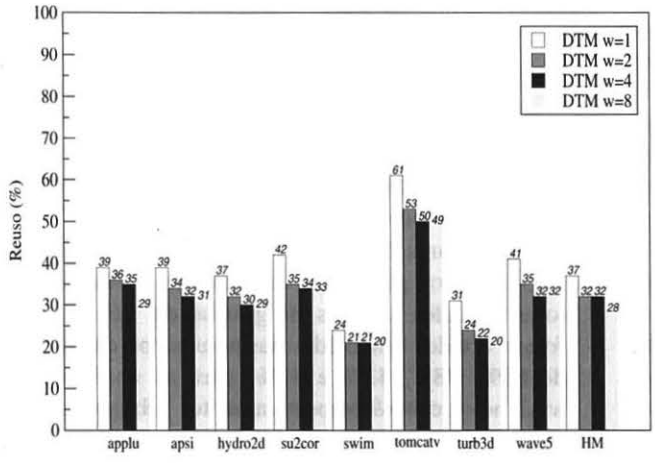

Figura 6. Reuso explorado pelo *DTM* em processadores com diferentes larguras, *SPECFp95*

B. Ganhos de Performance

Considerando os resultados expostos pela Figura 7, para os programas do *SPECInt95* foram obtidos em média (harmônica), ganhos de performance de 25%, 19%, 8.4% e 6.3% para as configurações *w=1*, *w=2*, *w=4* e *w=8* 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 o programa *li*, a configuração *w=8* apresentou ganhos de performance (16%) maiores que a configuração *w=4*. Para a maioria dos programas, a configuração *w=1* apresentou os maiores ganhos, seguida da configuração *w=2*, *w=4* e *w=8* (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 traços redundantes. Esta característica gerou investigações que constataram um significativo aumento no número de traços que foram reusados quando as larguras são reduzidas. Os efeitos produzidos nos ganhos de performance pelo reuso de traços, são distintos para as diferentes configurações avaliadas. Por exemplo, reusando traços 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 as configurações *w=4* e *w=8*, é evitada apenas 1 (uma) execução (desconsiderando para todos os casos, as limitação de unidades funcionais disponíveis).

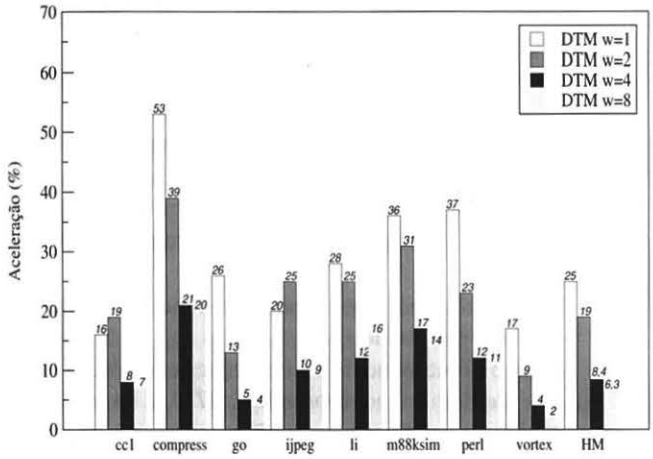

Figura 7. Ganhos de performance obtidos pelo *DTM* em processadores com diferentes larguras, *SPECInt95*.

Considerando os resultados expostos pela Figura 8, para os programas do *SPECFp95* foram obtidos em média (harmônica), ganhos de performance de 9%, 13%, 7% e 6% para as configurações *w=1*, *w=2*, *w=4* e *w=8* respectivamente. A configuração *w=2* apresentou um maior ga-

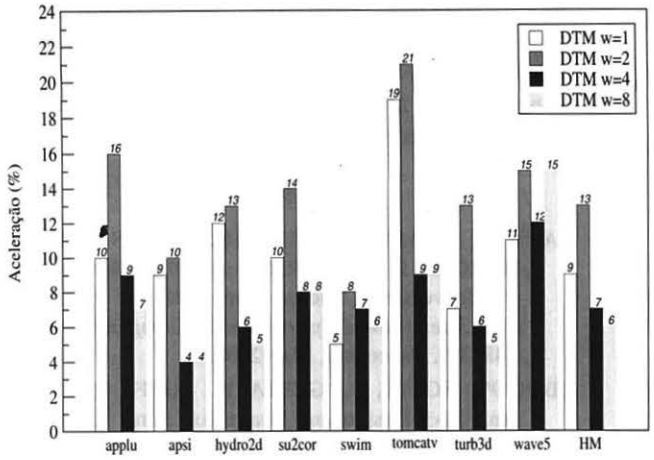

Figura 8. Ganhos de performance obtidos pelo *DTM* em processadores com diferentes larguras, *SPECFp95*.

nho de performance, seguida em ordem decrescente pelas configurações *w=1*, *w=4* e *w=8*. Excessões ocorreram para os programas *swim* e *wave5*, nestes, o ganho de performance da configuração *w=4* e *w=8* respectivamente, foram 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*.

C. Análise Comparativa

As medidas a seguir, plotam os valores de *ipc* obtidos para as configurações *w=1 base*, *w=2 base*, *w=4 base* e *w=8 base*, considerando o processador substrato como base e, *w=1 DTM*, *w=2 DTM*, *w=4 DTM* e *w=8 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 9 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*. Observa-se ainda que, a configuração *w=4 DTM* apresenta para quase todos os programas (com excessão do vortex), valores de *ipc* superiores aos obtidos pela configuração *w=8 base*. A média harmônica obtida entre os valores de *ipc* para cada configuração revelaram que comparativamente: a configuração *w=2 base* apresentou um ganho de performance de 27% sobre a configuração *w=1 DTM*; a configuração *w=4 base* apresentou um ganho de performance de apenas 8.6% sobre a configuração *w=2 DTM* e a configuração *w=4 DTM* apresentou um ganho de performance de 3.2% sobre a configuração *w=8 base*.

A Figura 10 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=2 base* apresentou um ganho de performance de 87% sobre a configuração *w=1 DTM*; a configuração *w=4 base* apresentou um ganho de performance de apenas 2.5% sobre a configuração *w=2 DTM* e a configuração *w=4 DTM* apresentou um ganho de performance de 4.3% sobre a configuração *w=8 base*.

É imediato observar que a configuração *w=2 DTM* consegue obter de forma equilibrada com relação à configuração *w=4 base*, valores de reuso e ganhos de performance bastante atraentes, principalmente com relação aos programas do *SPECFp95*. A configuração *w=4 DTM* apresenta-se como uma alternativa real à estratégia de aumento da largura do processador (para este caso, de 4 para 8). Os valores obtidos para as configurações avaliadas fornecem uma base comparativa, que podem determinar alternativas de projeto (dependendo da aplicação) para processadores, visando a obtenção de um maior *ipc* e reduções de complexidade [PAL 96, AGA 00].

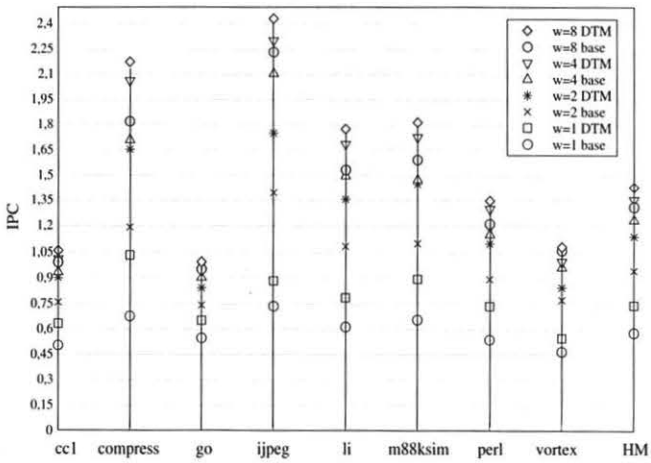

Figura 9. Valores de *ipc* para processadores incorporando o *DTM* e com diferentes larguras, *SPECInt95*.

VII. Conclusões e Trabalhos Futuros

As avaliações efetuadas, constataram que o mecanismo *Dynamic Trace Memoization* (*DTM*) incorporado a um processador superescalar considerando diferentes larguras, apresenta expressivos valores percentuais de reuso, e ganhos de performance ao explorar o reuso identificado. Considerando o conjunto de todas as configurações avaliadas, foram observados para o *SPECInt95*, percentuais de reuso de 60%, 48%, 43% e 41%, para as configurações do processador com lar-

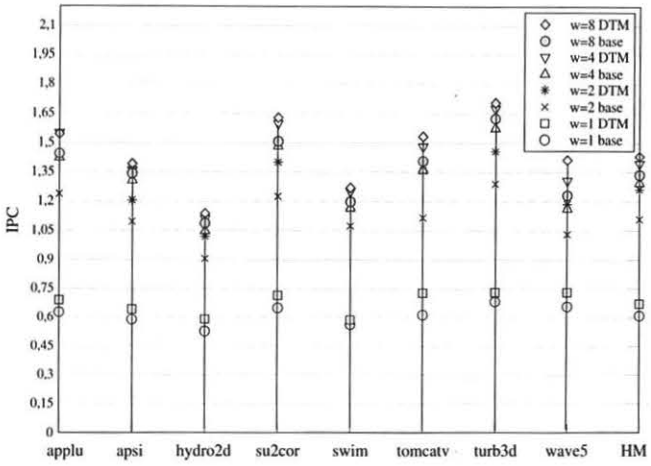

Figura 10. Valores de *ipc* para processadores incorporando o *DTM* e com diferentes larguras, *SPECFp95*.

gura 1, 2, 4, 8 respectivamente, e ganhos de performance de 25%, 19%, 8.4% e 6.3%, para as larguras 1, 2, 4, 8 respectivamente. Para o *SPECFp95* foram observados percentuais de reuso de 37%, 32%, 32% e 28%, para as larguras 1, 2, 4, 8 respectivamente, e ganhos de performance de 9%, 13%, 7% e 6%, para as larguras 1, 2, 4, 8 respectivamente. Estes resultados conferem a efetividade do mecanismo *DTM* para as diferentes configurações avaliadas.

Aliado aos valores previamente obtidos, a comparação de performance entre as diferentes configurações do processador incorporando o mecanismo *DTM* e as mesmas configurações do processador base (ambas considerando o processador configurado com diferentes larguras), estabeleceram valores que podem influenciar na estratégia de se aumentar as larguras (replicação dos estágios) dos processadores superescalares, para obtenção de um maior valor de *ipc* explorando apenas o paralelismo de instruções. Os resultados atestaram que: o processador superescalar base com largura 4 apresentou ganhos de performance de apenas 8.6% e 2.5% para o *SPECInt95* e *SPECFp95* respectivamente, sobre o mesmo processador superescalar com largura 2 e incorporando o mecanismo *DTM*; o processador superescalar com largura 4 incorporando o mecanismo *DTM*, produziu ganhos de performance em média harmônica de 3.2% e 4.3%, para o *SPECInt95* e *SPECFp95* respectivamente, sobre o mesmo processador superescalar base com largura 8. Estes resultados fornecem fortes indícios de que a exploração de redundância em nível de traços, apresenta-se potencialmente, como uma alternativa viável à estratégia de se aumentar as larguras (replicação dos estágios) dos processadores superescalares.

Finalmente, os trabalhos futuros serão direcionados a aperfeiçoamentos do mecanismo *DTM*: inclusão de instruções de memória aos traços considerados pelo *DTM*, de modo a aumentar o número de instruções por traço e produzir uma maior efetividade destes; o suporte do compilador para identificar prováveis invariantes do código gerado [BOD 99] e avaliação do *DTM* com relação à redução da potência consumida em decorrência do reuso.

Referências

[AGA 00] 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.

[BOD 99] 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.

[BUR 97] 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.

[COS 99] 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.

[COS 00] 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.

[GON 99] GONZALEZ A., TUBELLA, J., MOLINA, C. "Trace-Level Reuse". In: *Proceedings of the International Conference on Parallel Processing*, pp. 30-37, Japan, September 1999.

[HUA 99] 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.

[HUA 00] 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.

[LIP 96] 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.

[PAL 96] 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.

[SOD 97] SODANI, A., SOHI, G.S. "Dynamic Instruction Reuse". In: *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194-205, Denver, July 1997.

[SHI 98] SHRIVER, B., SMITH, B., *The Anatomy of a High-Performance Microprocessor: A Systems Perspective*. 1 ed. Los Alamitos, CA, IEEE Computer Society Press, 1998.