Reuso de Traços com Loads em Arquiteturas Superescalares*

Luiz S. Laurino, Tatiana S. G. dos Santos, Philippe O. A. Navaux
Universidade Federal do Rio Grande do Sul
{lslaurino, tatiana, navaux}@inf.ufrgs.br

Maurício L. Pilla
Universidade Católica de Pelotas
pilla@ucpel.tche.br

Resumo

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

A arquitetura RST original, no entanto, não considera instruções de acesso à memória como candidatas ao reuso. Desse modo, esse trabalho tem como principal objetivo analisar o impacto causado pela inclusão de tais instruções no domínio de reuso da arquitetura. São apresentados resultados da composição dos traços e de speedup alcançados pelo mecanismo proposto. Simulações com previsões perfeitas e diferentes políticas para a formação dos traços são mostradas, a fim de determinar o desempenho alcançado pela arquitetura, bem como validar o mecanismo em si.

1. Introdução

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

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

O número de unidades funcionais no estágio de execução varia de acordo com cada processador, mas já é comum encontrar oito ou mais unidades funcionais disponíveis. Mesmo contando com grande quantidade de hardware, o número de instruções executadas a cada ciclo (*Instructions Per Cycle* ou IPC) é baixo. Tipicamente, processadores do estado da arte não atingem, em média, IPC igual a 2 [10].

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

A maioria dos mecanismos desenvolvidos para aumentar o desempenho de processadores superescalares foca sua atenção na busca por paralelismo em nível de instrução (ILP, em inglês). Entretanto, tais mecanismos não levam em conta a redundância naturalmente encontrada em programas, que acontece pelo fato de que grande parte deles são genéricos e, portanto, apresentam uma série de laços e sub-rotinas, o que faz com que uma parcela significativa de código seja re-executada.

Estudos indicam que programas são constituídos de uma grande quantidade de computação redundante e previsível [7, 3, 19, 17]. Tal característica provocou o desenvolvimento de várias técnicas [13, 4, 20, 15] baseadas no con-

* Candidato ao Prêmio WSCAD 2005 de Melhor Artigo Completo de Estudante

ceito de localidade de valores [14] para ganhar desempenho através da exploração da redundância encontrada na execução de programas.

Em uma dessas técnicas, chamada RST (*Reuse through Speculation on Traces*) [15], é proposto o uso de duas metodologias simultaneamente: reuso e previsão de valores. Tal proposta prevê o uso conjunto das duas abordagens, de maneira a não se ter grande acréscimo de hardware se comparado a outras técnicas que utilizam as duas abordagens de forma isolada. O desenvolvimento deste mecanismo foi baseado no *Dynamic Trace Memoization* (DTM) [4] onde é realizado apenas o reuso de traços, não sendo considerada a execução especulativa.

A atual implementação da arquitetura RST, no entanto, não considera instruções de acesso à memória como parte do domínio de reuso. Instruções que acessam a memória necessitam de um tratamento especial para serem reusadas, o que não estava previsto na arquitetura RST original. Desse modo, o objetivo principal desse trabalho é analisar o impacto que o reuso de instruções de acesso à memória causa na arquitetura RST. Nesse artigo são apresentados resultados quantitativos dos limites superiores alcançados pela introdução de acessos à memória ao domínio de reuso nessa arquitetura.

Na próxima Seção são apresentados trabalhos relacionados, enquanto que a Seção 3 mostra a arquitetura RST. Na Seção 4 o mecanismo para incluir instruções de acesso à memória no domínio de reuso da arquitetura RST é apresentado. Na Seção 5 é descrito o ambiente de simulação. Os resultados e limites do mecanismo são fornecidos na Seção 6. Por fim, apresentamos algumas considerações finais e trabalhos futuros na Seção 7.

2. Trabalhos Relacionados

2.1. Reuso de Valores

O reuso de valores é uma forma não especulativa de explorar a redundância encontrada em diversas sequências de execução. Trata-se de um mecanismo que utiliza as entradas e saídas previamente armazenadas de uma determinada computação para evitar a execução de tarefas redundantes [19].

A medida que uma computação é executada, suas entradas e saídas são guardadas em uma tabela indexada, geralmente, pelo PC (*Program Counter*). Na próxima vez que esta computação for executada, seu contexto de entrada é comparado com aquele armazenado na tabela e caso sejam os mesmos valores, a saída previamente armazenada na tabela é copiada diretamente para os registradores de saída. Dessa forma, tem-se uma economia de recursos importante, visto que alguns estágios do pipeline não são utilizados.

Diversos mecanismos de reuso foram propostos ao longo do tempo. Basicamente, o que diferencia as diversas abordagens é a granularidade de reuso. Existem mecanismos especializados em reuso de instruções [19], reuso de blocos básicos [11], bem como reuso de traços de instruções [5, 9].

Além disso, o reuso de valores pode aumentar o desempenho dos processadores da seguinte maneira: *(i)* instruções reusadas não são executadas, proporcionando economia de recursos; *(ii)* os resultados ficam disponíveis mais cedo; *(iii)* as dependências de dados são minimizadas, pois instruções dependentes entre si podem executar em paralelo.

A principal desvantagem do reuso de valores é a necessidade de espera por todos os operandos de entrada de uma instrução. Apenas quando todos estão disponíveis é que se pode fazer um teste por reuso e determinar se os valores serão reusados ou não. Portanto, em alguns casos, operandos que poderiam ser reusados não o são, pelo fato de não estarem disponíveis no momento do teste.

2.2. Previsão de Valores

A previsão de valores é uma técnica especulativa para aumentar o desempenho do processador através da execução antecipada de instruções redundantes presentes nos programas. Este mecanismo consiste em prever valores de determinados operandos com base em seus valores anteriores [13]. Ao contrário do reuso de valores, que é um mecanismo puramente não especulativo, esta técnica permite a execução de instruções sem que todos os seus operandos estejam disponíveis. Entretanto, como nem todas as previsões realizadas estarão corretas, devem existir mecanismos de recuperação do contexto correspondente ao instante imediatamente anterior àquele em que ocorreu o erro de previsão. Nesse caso, as instruções devem ser re-executadas com os operandos corretos.

Os mecanismos de recuperação já estão disponíveis em processadores superescalares pois é necessário reconstituir o contexto anterior em caso de desvios condicionais mal tomados. Nesse caso, com algumas pequenas modificações, estes mecanismos podem também ser utilizados pela técnica de previsão de valores.

As principais vantagens apresentadas pela previsão de valores podem ser assim relacionadas: *(i)* minimização de dependências de dados verdadeiras [17], pelo fato de permitir que instruções cujos operandos ainda não estão disponíveis sejam executadas; *(ii)* início da execução de instruções de forma antecipada; e *(iii)* redução da latência de instruções de acesso à memória.

Já a principal desvantagem da previsão de valores são as perdas causadas quando um operando é previsto erroneamente, pois exige-se que o contexto anterior ao erro seja restabelecido e as instruções após este ponto, re-executadas. Quanto mais certeza se tiver a respeito da previsão de valo-

res feita, menos penalidades serão impostas ao processador e mais desempenho poderá ser obtido.

Diversos estudos apontam a eficiência dessa técnica [8, 7, 14], mas nenhuma delas alia a previsão de valores com o reuso. A arquitetura RST [15], por sua vez, tem como principal objetivo unir essas duas técnicas e conseguir um desempenho superior ao atingido pelas duas técnicas separadamente. A próxima Seção apresenta a arquitetura RST.

3. A Arquitetura RST

A arquitetura RST utiliza reuso numa granularidade mais elevada, em nível de traço. Isso significa que, ao invés de reusar uma instrução apenas, o mecanismo tenta reutilizar várias instruções de uma só vez. Além disso, os operandos que formam um traço são previstos, caso os mesmos não estejam prontos para serem reusados em um dado momento. Assim, o reuso ocorre especulativamente, aumentando o número de traços reusados e, consequentemente, aumentando significativamente o desempenho em relação a um mecanismo de reuso de traços tradicional [15].

Na arquitetura RST os traços são dinamicamente construídos a partir de sequências de instruções redundantes e armazenados nas tabelas *Memo_Table_G* (tabela de memorização global, voltada para reuso de instruções) e *Memo_Table_T* (tabela de memorização de traços). A Figura 1 mostra uma entrada típica para a *Memo_Table_T*, sem suporte a reuso de instruções que acessam a memória no interior dos traços. O campo *pc* indica o endereço da primeira instrução do traço. Em seguida, a entrada da tabela possui o endereço da instrução (*npc*) imediatamente posterior ao traço. Os campos que seguem dizem respeito ao contexto de entrada (*icr* e *icv*) e ao contexto de saída (*ocr* e *ocv*). Os dois últimos campos são usados quando há desvios dentro do traço.

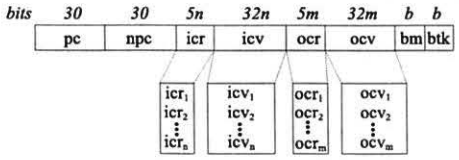

Figura 1. Exemplo de entrada da *Memo_Table_T*

Os traços podem ser constituídos de duas maneiras: (*i*) modo *reused-only*, quando somente instruções presentes na *Memo_Table_G* podem fazer parte de um traço e, (*ii*) modo *reusable*, quando instruções que não fazem parte da *Memo_Table_G* podem ser incluídas em um traço. A primeira política é a forma original de formação de traços (utilizada no DTM), enquanto que a segunda é uma forma um pouco mais agressiva de exploração de reuso.

A Figura 2 demonstra como é feita a formação dos traços utilizando a política *reusable*, que pode ser assim descrita: (*a*) a medida que instruções pertencentes ao domínio de reuso são encontradas, elas vão sendo armazenadas na *Memo_Table_T* até que uma instrução não pertencente ao domínio de reuso ou não redundante seja encontrada; (*b*) quando uma próxima execução alcança este traço, é feito um teste por reuso e se as entradas são as mesmas, as saídas são armazenadas nos registradores de saída. Caso algumas entradas sejam as mesmas e outras estejam sendo aguardadas, é feita uma especulação com relação às entradas não disponíveis e as saídas são também atualizadas. Após isso, o endereço de busca passa a ser aquele imediatamente posterior ao do traço.

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

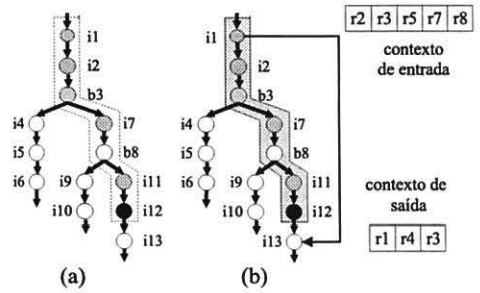

Figura 2. Formação dos traços

4. Acesso à Memória na Arquitetura RST

Instruções que envolvem a memória são largamente encontradas durante a execução de um programa. Sua inclusão no domínio de reuso do RST tornará os traços dinâmicos mais longos bem como minimizará o problema da indisponibilidade dos operandos de entrada e, potencialmente, ganhos serão alcançados.

Experimentos realizados em [15] (Figura 3) mostram que, para todos os traços criados (ou seja, aqueles que são armazenados na *Memo_Table_T* ou descartados porque as instruções não estão na *Memo_Table_G*), as instruções de acesso à memória são responsáveis por 45% da finalização de traços. Considerando apenas aqueles traços que estão armazenados na tabela de traços, as instruções de acesso à

memória representam quase que a totalidade das causas de finalização de traços.

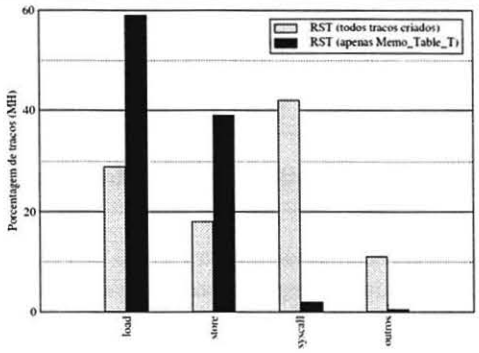

Figura 3. Principal causa de finalização dos traços no RST

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

Uma das principais preocupações quando instruções de acesso à memória são tratadas é quando uma instrução de escrita externa ao traço altera o valor da posição de memória referenciada por uma instrução dentro de um traço. Duas soluções podem ser implementadas [20]. A primeira alternativa, é invalidar os valores de operações de leitura nas tabelas *Memo_Table_G* e *Memo_Table_T*, interceptando todas as operações de escrita na memória e verificando se há correspondência entre o endereço a ser gravado na memória e aquele que é alvo de uma instrução de leitura presente em uma das tabelas acima mencionadas. Caso haja tal correspondência, um bit para valor de memória válido é desabilitado. Desta forma, quando é feito o despacho, este bit é consultado e, caso esteja desabilitado, é feita a execução da operação de leitura à memória; caso contrário, a instrução seguirá o fluxo de dados normal do mecanismo de reuso.

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

A implementação de instruções de *load* no RST trará custos adicionais ao mecanismo original. Entre estes custos destaca-se o aumento no tamanho das entradas das tabelas de reuso. Este aumento no tamanho depende do número de instruções de acesso à memória permitidas em cada traço.

O problema do aumento do tamanho das entradas pode ser contornado de diversas maneiras. A primeira diz respeito ao compartilhamento dos campos da área dedicada ao contexto de entrada e saída com os endereços das instruções de acesso à memória e os valores de retorno. Esta abordagem, no entanto, diminui a quantidade de operandos de entrada e saída disponíveis para um traço, causando diminuição no tamanho médio dos mesmos.

pc	sv1	sv2	maddr	mem val	res
30b	32b	32b	32b	1b	32b

Figura 4. Exemplo de uma entrada na *Memo_Table_L*

A segunda maneira de contornar o problema é através da criação de uma terceira tabela, chamada *Memo_Table_L*, responsável por armazenar instruções de leitura à memória que têm potencial para ser reusadas. Desse modo, *loads* não seriam enviados para a *Memo_Table_G* nem adicionados à tabela de traços. A Figura 4 mostra uma possível configuração para as entradas da *Memo_Table_L*, que teria um número de entradas bem menor do que as duas anteriores.

Como terceira alternativa, o mecanismo pode simplesmente não armazenar o valor de um *load*, indicando em uma tabela se os valores de memória são válidos. Portanto, esta tabela seria endereçada pelos endereços dos *loads* presentes em um traço e teria um bit associado a cada entrada, indicando se o valor é válido ou não. O Intel Itanium utiliza essa estratégia para *loads* especulativos através da *Advanced Load Address Table* (ALAT) [18, 12]. Neste caso, a posição da tabela que faz referência a um endereço de memória é invalidada caso ocorra um *store* entre o *load* especulativo e o *load* na sua posição original (não-especulativo). Outros trabalhos desenvolvidos previamente também utilizam mecanismos similares para controle de predicação e execução de múltiplos fluxos [6].

Finalmente, uma quarta alternativa diz respeito à utilização de reuso perfeito de acessos à memória, a fim de determinar os limites de ganho para esse mecanismo. A idéia do reuso é idêntica, mas ao invés de implementar um mecanismo dedicado para a reutilização de *loads*, a arquitetura simplesmente testa se o valor armazenado na memória ainda é o mesmo. Dessa forma, o armazenamento do valor de memória em uma tabela de reuso passa a ser desnecessário. É possível notar que, para as demais instruções, o mecanismo de reuso atua normalmente, ou seja, esse mecanismo é perfeito apenas para acessos à memória. Nesse trabalho, esta é a abordagem que está sendo utilizada, visto que o objetivo principal é estabele-

cer o impacto causado pelo emprego de um mecanismo de reuso de instruções de carga.

5. Ambiente de Simulação

Para as simulações deste trabalho, o simulador *sim-rst* desenvolvido em [15] e baseado no SimpleScalar Tool Set [2] foi utilizado. Esse simulador implementa um processador superescalar com profundidade de estágios no *pipeline* configurável e com o mecanismo de reuso especulativo de traços (RST).

Arquitetura	Suporte a *loads*	Política
DTM	sem suporte	*reused-only*
	com suporte	*reused-only*
	sem suporte	*reusable*
	com suporte	*reusable*
RST	sem suporte	*reused-only*
	com suporte	*reused-only*
	sem suporte	*reusable*
	com suporte	*reusable*

Tabela 1. Arquiteturas utilizadas nas simulações

A configuração utilizada foi baseada em processadores superescalares comerciais, com 20 estágios no *pipeline*, 128 entradas na lógica de reordenamento, 64 entradas na fila de *loads/stores*, três níveis de *cache* (64KB, 512KB e 2 MB) com tempos de acesso de 1, 5 e 20 ciclos, respectivamente, memória com tempo de acesso de 200 ciclos, mecanismo de previsão de desvios com dois níveis e 8KB, e BTB com 4096 entradas.

Além disso, o mecanismo foi simulado com as arquiteturas DTM (com reuso de valores) e RST (com reuso e previsão de valores), com e sem suporte a *loads* e com duas políticas de formação de traços: *reused-only* (traços não especulativos) e *reusable* (traços especulativos). Essas configurações são mostradas na Tabela 1.

As tabelas de reuso foram configuradas como mostrado na Tabela 2.

Adicionalmente, quando previsão de valores (reuso especulativo) é empregada, até dois valores podem ser previstos por traço.

Apesar de estudos anteriores [16] mostrarem que para a arquitetura RST e *pipelines* profundos os melhores resultados são obtidos com três registradores no escopo de entrada e dois na saída, neste estudo optou-se pelo uso de um número maior de registradores para formação dos traços. Com a inclusão de instruções de *loads*, os traços tendem a ser significativamente maiores. Desse modo, o aumento do limite máximo de registradores nos escopos de entrada e de saída permite que mais traços sejam formados e, eventualmente, reusados.

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

Tabela 2. Configuração das tabelas de reuso

As simulações foram feitas utilizando programas e entradas do SPEC 95int (*cc1*, *compress*, *go*, *ijpeg*, *li*, *perl*, *vortex*) e SPEC 2000int (*cc1*, *gzip*, *parser*, *vortex*) [1]. *Benchmarks* de inteiros foram escolhidos pelo fato de o RST não reusar instruções de ponto flutuante. Todos *benchmarks* executaram no máximo 1 bilhão de instruções que foram efetivadas ou até o seu final.

6. Resultados

Esta seção apresenta os resultados obtidos pelas simulações realizadas com o mecanismo proposto. Num primeiro momento, é apresentada uma análise quanto aos *speedups* encontrados. Na seção seguinte, é feita uma discussão acerca de diversas características dos traços.

6.1. Speedups

A Figura 5 mostra o *speedup* sobre a arquitetura base (sem reuso) para quatro configurações de reuso não-especulativo (abordagem utilizada pelo DTM). O eixo vertical da Figura apresenta o *speedup* alcançado, enquanto que o eixo horizontal mostra os *benchmarks* simulados. O primeiro conjunto de barras (DTM) apresenta os resultados de DTM sem reuso de memória e com a técnica de construção de traços original. O segundo conjunto, DTM + mem, mostra os resultados obtidos com a adição de acessos à memória no conjunto de instruções reusáveis. Os dois conjuntos de barras que seguem apresentam os resultados para as mesmas configurações apenas trocando a técnica de construção de traços.

O reuso de instruções de memória melhorou o desempenho em praticamente todos os casos, aumentando o *speedup* médio (média harmônica) em 3%, de $1,23$ para $1,26$, no caso da construção de traços original, e de $1,19$ para $1,22$, no caso da construção de traços especulativos. Para casos específicos como o *benchmark Perl*, a diferença a favor do mecanismo com reuso de memória é superior a 5%

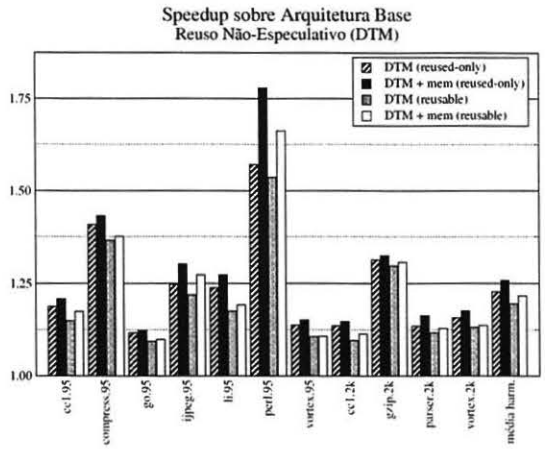

Figura 5. *Speedup* sobre arquitetura sem reuso para o DTM

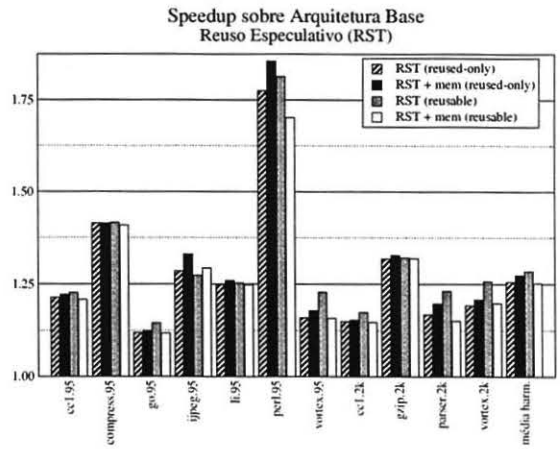

Figura 6. *Speedup* sobre arquitetura sem reuso para o RST

no caso da formação original dos traços, sem traços especulativos. Como já determinado em [15], a melhor técnica de construção de traços para reuso não-especulativo segue sendo a original, determinada por Costa [4].

A Figura 6 mostra o *speedup* para as mesmas configurações da Figura 5 mas com reuso especulativo de traços (arquitetura RST). Neste caso, o reuso de instruções de memória só mostrou-se vantajoso quando a política original para formação dos traços foi utilizada, tendo o *speedup* médio (média harmônica) passado de 1,26 para 1,27. Quando a política de formação de traços especulativos é utilizada, o *speedup* médio caiu de 1,28 para 1,25.

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

6.2. Composição dos traços

Após estudar o desempenho das diferentes opções para formação de traços, é importante relacionar esse desempenho com as características que os traços reusados possuem em cada uma das configurações. A Figura 7 mostra o comprimento médio dos traços reusados no DTM e no RST, com e sem suporte a *loads*. Além disso, para o RST, são mostrados resultados de traços com reuso, onde todos os operandos do contexto de entrada estão disponíveis e, também, de traços com previsão, onde até dois operandos são previstos. No gráfico, o eixo vertical mostra o número médio de instruções por traço, enquanto que o eixo horizontal mostra os *benchmarks* simulados.

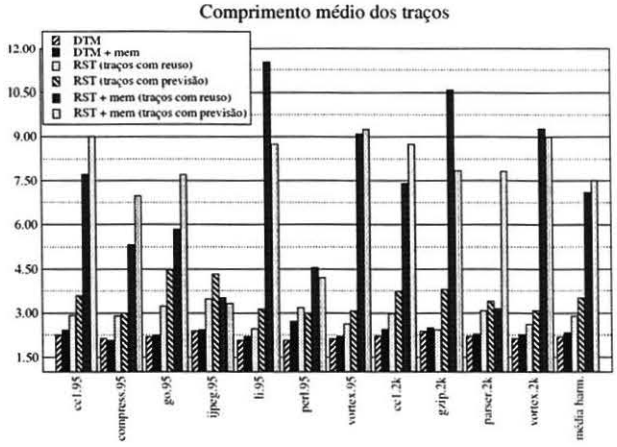

Figura 7. Número médio de instruções por traço

O comprimento médio dos traços aumentou de 3,5 no RST original [15] para cerca de 7 no RST com suporte a *loads*, como pode ser visto na Figura 7. De acordo com o que foi apresentado nas Figuras 5 e 6, uma das razões para a me-

lhoria de desempenho do mecanismo proposto em relação a arquitetura base é o fato de os traços apresentarem tamanho maior. Entretanto, o tamanho do traço simplesmente não implica em reuso e ganho de desempenho, visto que traços maiores tendem a apresentar um número de operandos de entrada maior, o que, potencialmente, pode significar menos redundância e menos reuso.

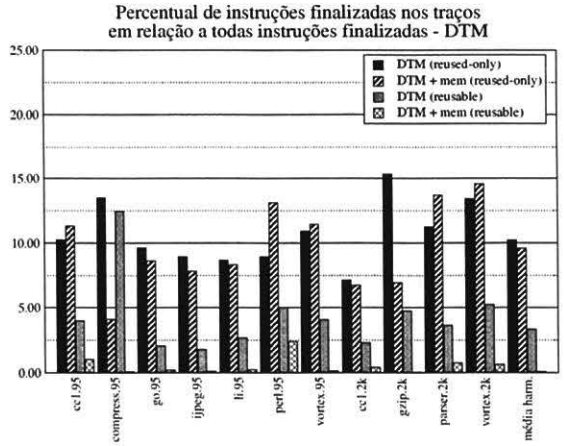

Figura 8. Percentual de instruções finalizadas nos traços no DTM

O ganho obtido em relação à arquitetura base pelo mecanismo proposto em termos de instruções não executadas e de acordo com o tipo de reuso (DTM e RST) pode ser visto nas Figuras 8 e 9. O eixo vertical mostra o percentual de instruções finalizadas nos traços reusados, enquanto que o eixo horizontal mostra os *benchmarks* simulados. As barras mostram resultados com e sem suporte a instruções de *load* e com as duas políticas de formação de traços. O RST, como mostrado em [15], alcança uma taxa de reuso maior do que o DTM pelo fato de utilizar reuso especulativo de traços. Entretanto, percebe-se que o número de instruções reusadas pelo RST com memória, em média, é inferior ao RST original, com ou sem traços especulativos. O fato é que, ao se fazer uma correlação com o *speedup* mostrado na Figura 6, pode-se concluir que, mesmo reusando um número menor de instruções, o RST com memória reusa instruções com latências maiores (como é o caso dos *loads*) e, portanto, o aumento de desempenho é justificado.

A Figura 10 apresenta o percentual de traços com a presença de instruções de *loads* que foram efetivamente reusados. O eixo vertical representa o percentual de traços reusados, enquanto o eixo horizontal mostra os *benchmarks* testados. Os dois primeiros conjuntos de barras referem-se

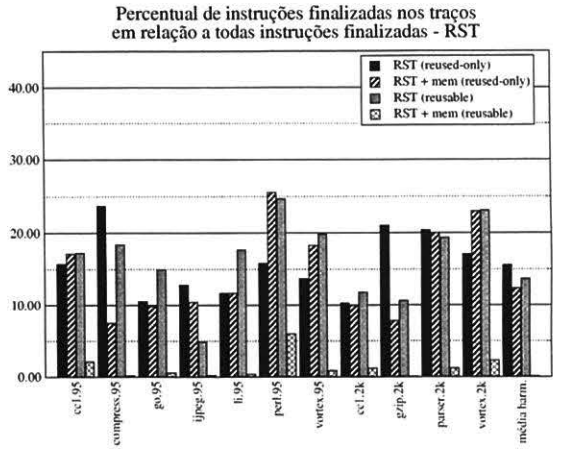

Figura 9. Percentual de instruções finalizadas nos traços no RST

ao DTM com as duas políticas de formação de traços. A seguir, o próximo conjunto é uma composição dos resultados dos traços com reuso e com previsão de valores, ambos com a política original. Por fim, o último conjunto de barras é similar ao anterior, porém com a outra política de formação de traços. É possível verificar que, para todos os casos, o número de traços com *loads* que foram reusados no DTM é inferior aos valores encontrados no RST. Além disso, na arquitetura RST, para os *benchmarks cc1.95*, *go.95*, *perl.95*, *cc1.2k* e *parser.2k*, entre 40 e 60% dos traços reusados, em média, possuíam instruções de carga na memória. Em casos mais específicos, como nos *benchmarks li*, *vortex.95* e *vortex.2k*, o percentual de traços reusados na arquitetura RST que possuíam *loads* ultrapassou 90%.

7. Considerações Finais

Este trabalho apresentou um estudo sobre o uso de instruções de acesso à memória no domínio de reuso da arquitetura RST. De acordo com os resultados obtidos, pode-se concluir que este mecanismo mostra-se vantajoso em relação ao antecessor. Entretanto, deve-se atentar para a política de formação dos traços, visto que esta afeta tanto o reuso especulativo como o não especulativo. Especialmente no RST com instruções de acesso à memória, a política original mostrou-se mais eficaz, como pode ser visto no *speedup* do *benchmark perl.95*, por exemplo, onde o ganho de desempenho foi de aproximadamente 5%.

Além disso, como mostrado nos resultados, percebe-se que o ganho de desempenho é obtido através da composição de dois fatores: número de instruções reusadas e suas res-

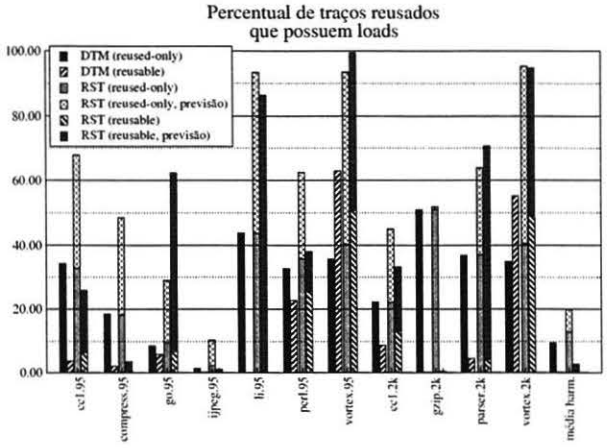

Figura 10. Percentual de traços reusados que contenham *loads*

pectivas latências. O RST com memória, mesmo apresentando, em média, $20,87\%$ menos instruções reusadas do que o RST original, possui desempenho 2% superior a este, visto que as latências das instruções reusadas pelo mecanismo proposto compensam o número menor de instruções reusadas. Em casos específicos, como no *benchmark ijpeg.95*, o número de instruções reusadas pelo RST com memória é $18,48\%$ menor do que o RST original, enquanto que o desempenho é cerca de $3,5\%$ superior.

Em trabalhos futuros pretende-se restringir o mecanismo, de modo que este passe a armazenar o valor de memória obtido por um *load*. Além disso, a utilização de *benchmarks* de ponto flutuante (SPEC FP) será avaliada a fim de determinar a possibilidade de ganho neste tipo de aplicação. Mesmo não apresentando muita redundância, esses *benchmarks* possuem instruções com altas latências de execução e, pelo que foi apresentado anteriormente, existe uma alta possibilidade de obtenção de ganhos de desempenho.

Agradecimentos

Os autores agradecem o suporte do CNPq na forma de concessão de bolsas de estudo.

Referências

[1] Standard performance evaluation corporation. http://www.spec.org.

[2] T. M. Austin and D. Burger. Simplescalar tutorial, 2001.

[3] R. Bodik, R. Gupta, and M. L. Soffa. Load-reuse analysis: Design and evaluation. In *Proc. of SIGPLAN Conference on Programming Language Design and Implementation*, pages 64–76. New York, ACM, 1999.

[4] A. T. da Costa. *Exploiting Dynamically the Reuse of Traces in Processor Architecture Level*. Phd thesis, COPPE-UFRJ, 2001.

[5] A. T. da Costa, F. M. G. França, and E. M. C. Filho. The dynamic trace memoization reuse technique. In *Proc. of the 9th International Conference on Parallel Architecture and Compiler Techniques*, pages 92–99, Philadelphia, Oct. 2000. Los Alamitos, IEEE Computer Society.

[6] R. R. dos Santos. *DCE: The Dynamic Conditional Execution in a Multipath Control Independent Architecture*. Phd thesis, II-UFRGS, May 2003.

[7] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. Technical Report EE Dept. #1080, Technion–Israel Institute of Technology, Israel, 1996.

[8] F. Gabbay and A. Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions on Computer Systems*, 16(3):234–270, 1998.

[9] A. González, J. Tubella, and C. Molina. Trace-level reuse. *International Conference on Parallel Processing*, pages 30–37, 1999.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 3rd ed. edition, 2003.

[11] J. Huang and D. J. Lilja. Exploring sub-block value reuse for superscalar processors. *2000 International Conference on Parallel Architecture and Compiler Techniques(PACT)*, 2000.

[12] Intel. *Itanium^{tm} Processor Microarchitecture Reference*. Intel Corporation, 2000.

[13] M. Lipasti. *Value Locality and Speculative Execution*. Phd thesis, Carnegie Mellon University, Apr. 1997.

[14] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. *ACM SIGPLAN Notices*, 31(9):138–147, 1996.

[15] M. L. Pilla. *RST: Reuse through Speculation on Traces*. Phd thesis, II-UFRGS, June 2004.

[16] M. L. Pilla, P. O. A. Navaux, B. R. Childers, A. T. da Costa, and F. M. G. Franca. Value predictors for reuse through speculation on traces. In *Proc. of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 48–55, Foz do Iguaçu, Oct. 2004. IEEE Computer Society.

[17] Y. Sazeides and J. E. Smith. The predictability of data values. *30th International Symposium on Microarchitecture*, pages 248–258, 1997.

[18] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, 2000.

[19] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *24th International Symposium on Computer Architecture(ISCA)*, pages 194–205, 1997.

[20] L. M. F. A. Viana. Memorização dinâmica de traces com reuso de valores de instruções de acesso à memória. Master's thesis, COPPE–UFRJ, Rio de Janeiro, Março 2002.