RSTm: Reusando Especulativamente Acessos à Memória

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

Tatiana G. S. dos Santos
Universidade de Santa Cruz do Sul
tatianas@unisc.br

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

Resumo

Técnicas de reuso e previsão de valores são alternativas para aumentar o desempenho em arquiteturas de processadores, já que permitem que instruções com dependências verdadeiras e de controle tenham seus resultados no mesmo ciclo. No entanto, arquiteturas que utilizam esses mecanismos não costumam incorporar os acessos à memória como parte das instruções que podem ser reusadas. Neste artigo, o RSTm é apresentado, uma versão da arquitetura RST (Reuse through Speculation on Traces) que permite o reuso (especulativo ou não) de acessos à memória. A verificação da reusabilidade de instruções de acesso à memória dá-se com o uso de uma tabela adicional, a Memo_Table_L, que armazena endereços e valores dos acessos reusáveis. Esta solução não limita o número de instruções de acesso à memória por traço e, também, armazena tanto o endereço como seu respectivo valor, com pequeno custo adicional no hardware. Os experimentos, realizados com benchmarks do SPEC2000int e SPEC2000fp mostram um speedup de até 1,0474 no desempenho do RSTm sobre o mecanismo original e de 1,2019 sobre a arquitetura base.

1. Introdução

A busca crescente por desempenho na área de microprocessadores é tipicamente limitada por três fatores principais: dependências de dados, dependências de controle e conflito de recursos [7]. Nas arquiteturas do estado da arte, entretanto, onde milhares de transistores já estão disponíveis, os conflitos de recursos não costumam representar o pior gargalo. De fato, um maior *throughput* não é alcançado em virtude, principalmente, das dependências entre instruções.

As dependências de dados do tipo WAW (*Write After Write* ou dependências de saída) e WAR (*Write After Read* ou dependências falsas) são eficientemente tratadas por mecanismos tradicionais, como o Algoritmo de Tomasulo [19] e o *Scoreboard* [18]. Contudo, as dependências verdadeiras ainda são comumente encontradas e as técnicas disponíveis para tratá-las não são totalmente efetivas.

Outro problema grave e também sem solução definitiva são as dependências geradas pelas instruções de controle de fluxo. Estima-se que cerca de 20% da intruções, dentro de um programa qualquer, são de desvios. A ocorrência de qualquer erro de previsão pode ocasionar uma grande perda de desempenho. Em processadores modernos, com *pipelines* profundos, onde o número de estágios chega a 20, esse problema é bastante crítico e pode comprometer o desempenho global da arquitetura.

O esforço conjunto da indústria e do meio acadêmico tem gerado muitas pesquisas que visam sanar ou minimizar esses problemas. Dentro desse contexto, é possível destacar mecanismos que utilizam a localidade de valores [12] para ganhar desempenho através da exploração da redundância encontrada na execução de programas. Isso é possível pois a quantidade de computação redundante e previsível é grande.

Um desses mecanismos, o RST (*Reuse through Speculation on Traces*) [13], propõe o uso conjunto de duas técnicas simultaneamente: reuso e previsão de valores. A arquitetura que implementa o RST prevê o uso conjunto dessas duas abordagens, de maneira a não apresentar um grande acréscimo de hardware se comparado a outras arquiteturas que utilizam as duas abordagens de forma isolada.

A utilização do RST trouxe resultados bastante significativos, mas o domínio de reuso adotado originalmente não permitia o reuso/previsão de instruções de acesso à memória. Desse modo, esse trabalho tem como principal objetivo apresentar o RSTm (*Reuse through Speculation on Traces with Memory*), uma extensão do RST para incluir instruções de acesso à memória no seu respectivo domínio de reuso. Esse trabalho mostra que a inclusão dessas instruções pode efetivamente aumentar o desempenho da arquitetura base.

Na próxima Seção são apresentados trabalhos relaciona-

dos, 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

O **Reuso de Valores** é uma forma não especulativa de explorar a redundância encontrada em diversas seqüências 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 [16].

Conforme 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 [16], reuso de blocos básicos [8], bem como reuso de traços de instruções [3, 6].

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 são disponibilizados mais cedo; e *(iii)* as dependências de dados são minimizadas, pois instruções dependentes entre si podem executar em paralelo.

Jin e Cho [9] apresentam um estudo a respeito do reuso de valores de memória possíveis de serem explorados em programas comuns. Os estudos e análises apresentados pelos autores consideram seqüências genéricas de instruções, não levando em conta seqüências de instruções pertencentes a um traço, por exemplo. Utilizando uma tabela para reuso de valores de memória (MVRT ou *Memory Value Reuse Table*), os autores mostraram que aproximadamente 70% dos *loads* tiveram seus valores reusados (execuções com SPEC2000int e MiBench): *(i)* 20-25% dos *loads* obtiveram o valor reusado através de *stores* executados anteriormente e, *(ii)* entre 30-40% dos *loads* obtiveram seus valores devido à execução prévia de outras instruções de *load*. Já no SPEC2000fp, o potencial de reuso de *load*-para-*load* foi de 44%.

Bodík, Gupta e Soffa [2] produziram um estudo quanto ao limite de reuso de instruções de acesso à memória, além de desenvolverem uma técnica de avaliação de reuso de *loads* em nível de compilação. Os resultados por eles alcançados atingem 55% de exposição de reuso de *loads* em *benchmarks* do SPEC95, sendo que este valor chega a 80% quando a técnica de compilação é utilizada.

A principal desvantagem do reuso de valores é a necessidade de esperar até que todos os operandos de entrada de uma instrução estejam prontos. Apenas quando todos estão disponíveis é que é possível fazer um teste de reuso e determinar se esses valores podem ser reusados ou não. Desse modo, em alguns casos, operandos que estão presentes na tabela de reuso e poderiam ser reusados não o são, pelo fato de não estarem prontos no momento do teste.

A **Previsão de Valores**, por sua vez, 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 [11]. 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, faz-se necessário a existência de 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 também é necessário reconstituir o contexto anterior em caso de desvios condicionais previstos incorretamente. Esses mesmos mecanismos, com algumas pequenas modificações, 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 [15], 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.

Por outro lado, a principal desvantagem da previsão de valores são as perdas causadas quando um operando é previsto erroneamente. Assim como ocorre com desvios incorretos, o contexto anterior ao erro deve ser restabelecido e as instruções após este ponto, re-executadas. Quanto mais confiável for a previsão de valores, menos penalidades serão impostas ao processador e mais desempenho poderá ser obtido.

Diversos estudos apontam a eficiência da previsão de valores [5, 4, 12], mas sem considerar a previsão de valores

em conjunto com o reuso. A arquitetura RST [13], 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. A arquitetura RST difere da arquitetura que lhe deu origem, a DTM (*Dynamic Trace Memoization*), em virtude de oferecer o reuso com previsão. Assim, os operandos que formam um traço são previstos, caso os mesmos não estejam prontos para serem reusados em um dado momento. 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 [13].

Na arquitetura RST os traços são dinamicamente construídos a partir de seqüências 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.

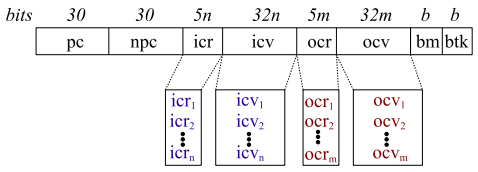

Figura 1. Exemplo de entrada da *Memo_Table_T*

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*) à 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, um teste de reuso é realizado 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 igualmente 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.

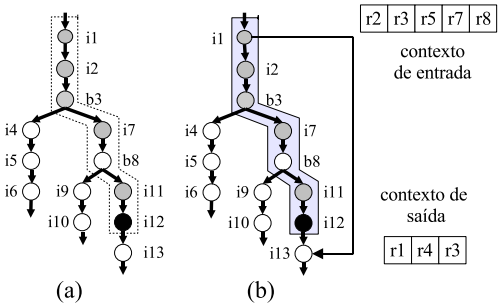

Figura 2. Formação dos traços

4. Acesso à Memória na Arquitetura RSTm

Instruções de acesso à memória podem envolver grandes latências e, portanto, são candidatas interessantes para técnicas que pretendam melhorar o desempenho de um processador. Em [10], o potencial de incluir acessos à memória no domínio de reuso da arquitetura RST foi estudado, demonstrando-se que o desempenho aumentava em torno de 2% quando comparado com a arquitetura sem reuso de acessos à memória. No entanto, esse estudo não considerou as restrições de implementação para as estruturas que armazenam e comparam os endereços da memória aos valores previamente encontrados. Outra questão em aberto referia-se ao efeito em *benchmarks* de ponto flutuante do reuso de acessos à memória.

A implementação de instruções de acesso à memória no RST resultou em um novo mecanismo chamado **RSTm (Reuse through Speculation on Traces with Memory)** e implicou em três significativas alterações no mecanismo original. A primeira modificação diz respeito à criação de uma terceira tabela de reuso chamada *Memo_Table_L* (Figura 3), responsável pelo armazenamento e controle dos valores lidos/escritos na memória por instruções de *load*/*store*, respectivamente. A nova tabela será responsável por armazenar dados das instruções de acesso à memória, terá poucas entradas em relação às outras tabelas (de traços e instruções) e seu conteúdo será sempre parte do contexto de saída de um traço. Além disso, para determinar se um traço possui alguma instrução de *load* ou *store*, basta verificar se o endereço da instrução está entre os limites do traço (`pc` e `npc`, campos presentes na Memo_Table_T). O inverso também pode ser facilmente determinado.

pc	sv1	type	maddr	valid	res
30b	32b	2b	32b	1b	32b

Figura 3. Exemplo de uma entrada na *Memo_Table_L*

Cada entrada na Memo_Table_L possui os seguintes campos:

- **pc**: o endereço da instrução de *load*;
- **sv1**: valor do operando fonte;
- **type**: mapa de bits que indica o tipo de acesso (*half, single, double*);
- **maddr**: armazena o endereço de acesso à memória;
- **valid**: *flag* que indica se o valor de leitura/escrita é válido;
- **res**: valor de leitura/escrita.

A fim de evitar o desperdício de espaço na Memo_Table_L, preferiu-se deixar o campo responsável pelo armazenamento do resultado da operação de memória (res) com 32 bits. Caso uma operação utilize um dado do tipo *double*, o campo `type` é configurado de forma apropriada e uma outra entrada na Memo_Table_L será alocada com o objetivo de armazenar o restante do valor. Portanto, um valor do tipo *double* fica armazenado em duas entradas da Memo_Table_L, como pode-se ver na Figura 4.

A segunda modificação, por sua vez, foi realizada no caminho percorrido pelos traços nos estágios do *pipeline* do RST, visto que agora os traços que possuem *stores* terão um comportamento levemente diferente do que normalmente

pc	sv1	type	maddr	valid	res
1000	R5	dbl	128	yes	AC
1000	R5	cont. dbl	12C	yes	9F

Figura 4. Exemplo de uma entrada na *Memo_Table_L* para um valor de 64 bits

era realizado. No caso de uma instrução de escrita gravar um valor diferente do último valor usado pela mesma, o *store* não poderá ser reusado e deverá ser normalmente executado (envio para a cache de dados). Além disso, caso a Memo_Table_L tenha um *load* apontando para o mesmo endereço que este *store*, a entrada da tabela correspondente a esta instrução de leitura deve ser invalidada (pelo menos até a instrução de escrita ser confirmada).

Finalmente, a terceira modificação significante foi realizada na fila de acesso à memória, onde instruções do tipo *store* são interceptadas a fim de determinar se há algum acesso a um mesmo endereço presente nas tabelas de reuso. Para tanto, o estágio RS4 sofreu uma pequena modificação no sentido de sinalizar ao RS1 que uma instrução de escrita foi terminada e que, caso exista alguma entrada na Memo_Table_L cujo endereço seja o mesmo da instrução que acaba de ser confirmada e não exista mais nenhuma instrução no *Reorder Buffer* com mesmo endereço-alvo, o mecanismo deve mudar o bit de validade da entrada da tabela para válido.

A Figura 5 mostra o *pipeline* incluindo o mecanismo RSTm. Na esquerda, é possível ver as três tabelas de reuso, Memo_Table_T, Memo_Table_G e Memo_Table_L. Os estágios do RSTm estão logo a seguir (RS1, RS2, RS3 e RS4). No estágio RS1, candidatos ao reuso convencional e especulativo são identificados nas tabelas de reuso e repassados ao estágio RS2. No estágio RS2, é feito o teste de reuso, comparando as entradas dos candidatos com os valores atuais no banco de registradores. Eventualmente, o mecanismo pode decidir por reusar especulativamente algum traço em que valores de entrada ainda não estão disponíveis. O estágio RS3 verifica e corrige previsões incorretas, redirecionando a busca como em um erro da previsão de desvios. Finalmente, o estágio RS4 é responsável por identificar instruções e traços com potencial para reuso e armazená-los nas tabelas de reuso.

5. Ambiente de Simulação

Para as simulações deste trabalho, o simulador *sim-rst* desenvolvido em [13, 14] e baseado no SimpleScalar Tool Set [1] foi utilizado como base, sofrendo as modificações

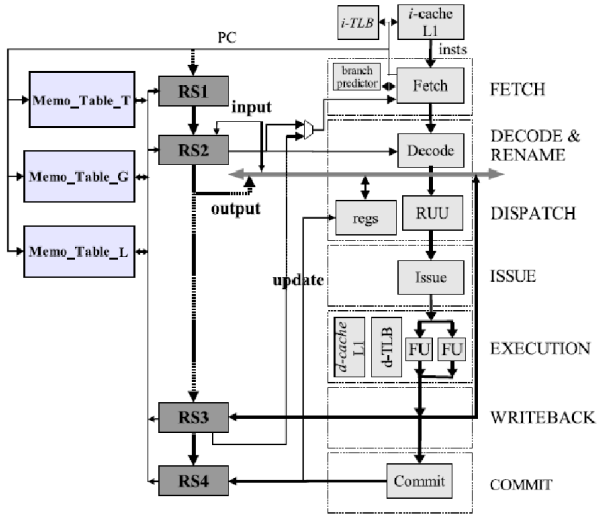

Figura 5. *Pipeline* incluindo o mecanismo RSTm

necessárias para modelar o comportamento de uma arquitetura com o mecanismo RSTm.

A configuração foi baseada em processadores superescalares comerciais, embora não represente nenhum processador em especial. O processador simulado tem *pipeline* de 20 estágios, 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. O tempo de acesso das *caches* foi configurado de modo a não aumentar as vantagens do RSTm (a tendência é que maiores latências tragam mais benefícios).

Os *benchmarks* utilizados são um subconjunto dos programas do SPEC2000int e SPEC2000fp, executados com as entradas de tamanho reduzido do [17] por um bilhão de instruções graduadas ou até o final da execução, o que ocorresse primeiro.

Resultados em [10] mostraram que o tamanho médio dos traços aumenta quando acessos à memória são incluídos em traços. Assim, optou-se por aumentar o contexto de entrada e saída (registradores que são usados pelo traço como entrada e saída) armazenados na tabela de traços Memo_Table_T. Assim, foram utilizados 8 registradores no contexto de entrada, mais 8 registradores no contexto de saída, ao invés dos 3 registradores na entrada e 2 na saída tidos como ideal no RST sem reuso de memória [13, 14]. As tabelas de reuso foram configuradas como mostrado na Tabela 1. Como as tabelas são acessadas em um *pipeline* separado, fora do caminho crítico, as tabelas de reuso podem ter latência de acesso maior que as *caches*, sem afetar seu desempenho. Desta forma, a variação no tempo de acesso às tabelas não foi considerado no presente estudo.

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
Memo_Table_L	Entradas	128
	Associatividade	4

Tabela 1. Configuração das tabelas de reuso

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. Depois, o tamanho dos traços é discutido e, finalmente, características dos traços.

6.1. Speedups

As Figuras 6 e 7 mostram o *speedup* sobre a arquitetura RST original (sem suporte à memória) e sobre a arquitetura base (sem reuso), respectivamente. O eixo vertical das Figuras apresenta o *speedup* alcançado, enquanto que o eixo horizontal mostra os *benchmarks* simulados. Na Figura 7, o primeiro conjunto de barras apresenta os resultados do RSTm com a política de construção de traços original (*reused-only*). O segundo conjunto mostra os resultados obtidos sobre o RST com a política de construção de traços mais agressiva, onde uma instrução não precisa ser reusada para ser candidata ao reuso (*reusable*).

Em comparação com o RST original, o reuso de instruções de memória melhorou o desempenho em praticamente todos os casos, com *speedup* médio (média harmônica) de $1,0474$ no caso da construção de traços original, e de $1,0217$, no caso da construção de traços de forma especulativa. Para casos específicos como o *benchmark gzip2.int*, o *speedup* do mecanismo com reuso de memória é de $1,84$ no caso da formação original dos traços, sem traços especulativos.

Ao comparar-se o mecanismo com a arquitetura base, os ganhos foram ainda maiores. O *speedup* para a construção original de traços foi de $1,2019$ em média, enquanto que o *speedup* médio para a construção especulativa

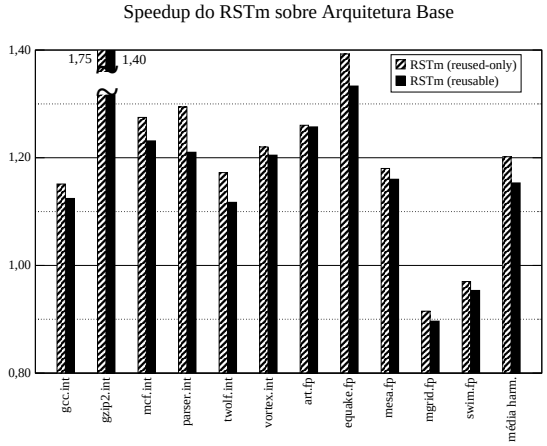

Figura 6. *Speedup* sobre arquitetura base

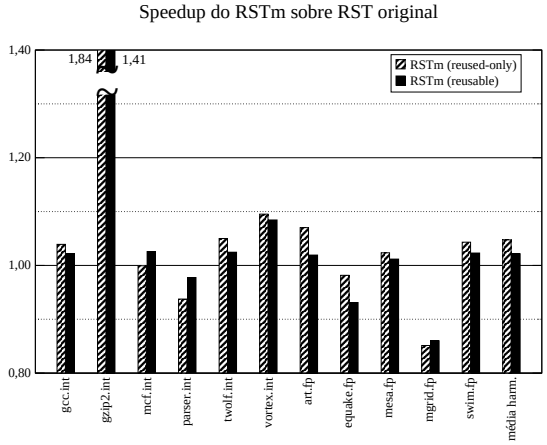

Figura 7. *Speedup* sobre arquitetura RST original

foi de $1,1532$. É interessante observar que seis *benchmarks* (*gzip2.int, mcf.int, parser.int, vortex.int, art.fp* e *equake.fp*) obtiveram *speedup* acima de $1,20$ para ambas as políticas de formação de traços.

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 pode 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.

Em [10], observou-se que os limites superiores de *speedup* para o RST com memória usando *bechmarks* do SPEC95 na sua maioria seria algo em torno de 2%. Entretanto, alguns *benchmarks* (*vortex.int*, por exemplo) apresentaram desempenho superior, por dois motivos:

- maior número de *loads* permitidos nos traços, bem como contexto de entrada e saída maiores do que nos experimentos realizados em [10];
- todos os *benchmarks* utilizados são do conjunto SPEC2000, o que pode ter contribuído para esta alteração no desempenho de alguns deles.

6.2. Tamanho Médio dos Traços

A Figura 8 mostra o tamanho médio dos traços reusados no DTM, RST e RSTm. Além disso, para o RST e RSTm, 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.

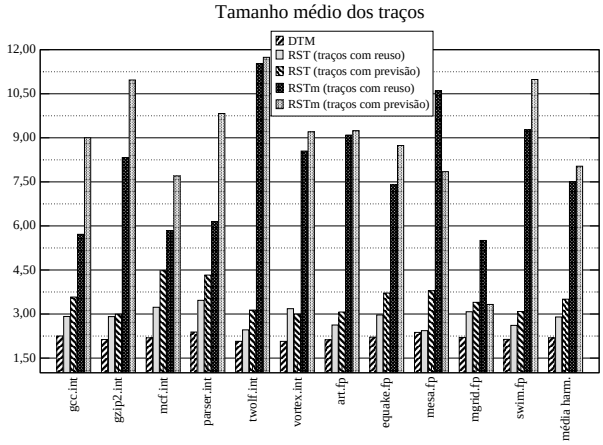

Figura 8. Número médio de instruções por traço

O tamanho médio dos traços aumentou de $3,1$ no RST original [13] para cerca de $7,5$ no RSTm. Uma das razões para a melhoria 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, poderia significar menos redundância e menos reuso.

6.3. Variação das Latências das Memórias

Quanto aos estudos relacionados ao impacto das latências das memórias no desempenho do RSTm, duas simulações distintas foram realizadas. Na primeira, a latência das memórias dos três primeiros níveis foi duplicada. Na segunda simulação, apenas a memória principal teve sua latência modificada de forma que seja o dobro daquela especificada na configuração original. A Tabela 6.3 mostra as latências para cada um dos níveis da hierarquia de memória.

Nível de Hierarquia	Lat. A	Lat. B
Primeiro Nível - Dados	2 ciclos	1 ciclo
Segundo Nível	10 ciclos	5 ciclos
Terceiro Nível	40 ciclos	20 ciclos
Memória Principal	200 ciclos	400 ciclos

Tabela 2. Latências das memórias para a configuração original

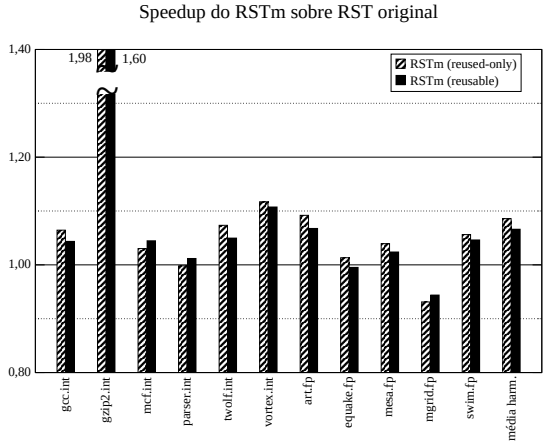

Figura 9. *Speedup* sobre o RST original (configuração A das memórias)

Após a execução dos *benchmarks* tendo as latências das memórias com a configuração A, o gráfico de *speedup* do RSTm em relação ao RST pode ser visto na Figura 9. Como esperado, o ganho de desempenho em relação ao RST original foi maior do que quando utilizando a configuração originalmente proposta, visto que o tempo de acesso a qualquer uma das memórias (exceto a memória principal) é maior na configuração A. Portanto, quando o mecanismo evita o acesso, a economia de tempo é maior neste caso e os ganhos médios ficaram em torno de $7,59\%$.

A Figura 10 mostra o *speedup* do RSTm em relação ao RST original para a configuração B. Neste caso, ocorre também um maior ganho de desempenho em relação àquele obtido com a configuração inicial. Entretanto, o ganho obtido não foi maior do que o medido no experimento anterior, tendo sido de $5,75\%$. Tal consideração atesta que a maior parte das instruções de acesso à memória reusadas são aquelas cujos dados de entrada estão armazenados nas *caches*.

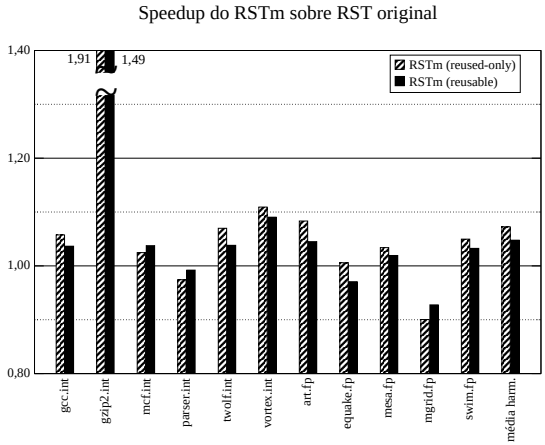

Figura 10. *Speedup* sobre o RST original (configuração B das memórias)

7. Considerações Finais

Neste trabalho, foi apresentado um mecanismo de reuso especulativo de traços com instruções de acesso à memória chamado RSTm (Reuse through Speculation on Traces with Memory). Esse mecanismo, baseado no RST, é implementado usando uma tabela adicional (Memo_Table_L) para manter referências à memória que poderiam ser reusadas.

A inclusão de instruções de acesso à memória ao mecanismo de reuso especulativo de traços em processadores superescalares fornece um *speedup* médio de $1,0474$ sobre o desempenho do RST, quando os traços são formados a partir de instruções reusadas anteriormente. Por outro lado, quando os traços são formados a partir de qualquer instrução executada o RST permite um *speedup* médio de $1,0217$, mostrando que traços com maior grau de redundância produzem melhores resultados no RSTm. Para alguns *benchmarks* em particular, no entanto, a diferença é bastante significativa, como no caso dos *benchmarks*

gzip2.int e *twolf.int*, onde a diferença de desempenho entre as duas formas de criação de traços foi de quase 100%.

Os resultados atestam que o ganho de desempenho é obtido através da composição de dois fatores: número de instruções reusadas e as suas respectivas latências. Mesmo apresentando 24% menos instruções reusadas do que o RST original, o RSTm consegue ser, em média, 2,97% mais rápido que aquele, já que as latências das instruções reusadas pelo mecanismo com memória compensam o menor número de instruções reusadas.

A maioria das instruções de acesso à memória reusadas pelo mecanismo possui seus dados em um dos três primeiros níveis de memória da arquitetura. Quando esses níveis têm sua latência aumentada, os *speedups* médios são de $1,0759$, ao passo que quando a memória principal tem sua latência aumentada, os ganhos ficam em torno de $1,0575$.

Como trabalhos futuros, pode-se citar a incorporação do reuso de memória à uma versão com tabelas de reuso unificada do RST, para diminuir os custos de implementação do mecanismo. Além disso, o RSTm poderia beneficiar-se de alterações nos compiladores para otimizar o uso do mecanismo de reuso.

Referências

[1] T. M. Austin e D. Burger. Simplescalar tutorial, 2001.

[2] R. Bodik, R. Gupta e M. L. Soffa. Load-Reuse Analysis: design and evaluation. SIGPLAN Conf. on Programming Language Design and Implementation, 1999, p.64–76.

[3] A. T. da Costa, F. M. G. França, e E. M. C. Filho. The dynamic trace memoization reuse technique. *Proc. of the 9th Intl. Conf. on Parallel Architecture and Compiler Techniques*, p.92–99, 2000.

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

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

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

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

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

[9] L. Jin and S. Cho. A Characterization Study on Memory Value Reuse. *Workshop on Memory Performance Issues – HPCA*, 2006.

[10] L. S. Laurino, T. S. G. dos Santos, P. O. A. Navaux e M. L. Pilla. Reuso de Traços com Loads em Arquiteturas Superescalares. *VI Workshop em Sistemas Computacionais de Alto Desempenho*, p.49–56, 2005.

[11] M. Lipasti. *Value Locality and Speculative Execution*. PhD Thesis, Carnegie Mellon University, Apr. 1997.

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

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

[14] M. L. Pilla, B. R. Childers, A. T. da Costa, F. M. G. França e P. O. A. Navaux. Limits for a Feasible Speculative Trace Reuse Implementation. *Intl. Journal of High Performance Systems Architecture*, 1(1):69-76, 2007.

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

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

[17] A. KleinOsowski, J. Flynn, N. Meares, e D. J. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-base computer architecture research. *Proc. of the Workshop for Workload Characterization – ICCD*, p.83–100, Austin, USA, 2000.

[18] J. E. Thornton. Parallel operation in the Control Data 6600. *AFIPS Fall Joint Computer Conference*, 26(2), 1964.

[19] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), Jan. 1967.